Miroslav Kubat

An Introduction to Machine Learning

Third Edition

Springer

An Introduction to Machine Learning

Miroslav Kubat

An Introduction to Machine Learning

Third Edition

Springer

Miroslav Kubat
Department of Electrical and Computer
Engineering
University of Miami
Coral Gables, FL, USA

ISBN 978-3-030-81934-7   ISBN 978-3-030-81935-4 (eBook)
https://doi.org/10.1007/978-3-030-81935-4

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my wife, Verunka

Acknowledgments

Writing a book is pleasant only at the beginning, when your heart overflows with energy and ambition. Soon, reality sets in. The progress is slower than you expected, and putting one word behind another is much harder than you once hoped. As the days stretch to weeks, then months, your determination slides away; the soaring spirits of the early days are gone. Somewhere deep inside, a nagging question burns: why did you commit yourself to this drudgery? So many projects had to be pushed aside, so many deadlines missed, so much time and energy invested—for a book that may yet prove a failure. And then, did your family deserve all those cancelled trips? The reader has got the message: moral support is important. Even the most independent mind will not go far without at least one fan.

Grateful thanks to my wife Verunka, for her encouragements and tolerance!

The book's contents owe a lot to my exposure to the hundreds of students that have passed through my machine learning classes. An old saw has it that a professor who wants to master a field proposes to teach it. There is an element of wisdom, in this. To develop a course, you have to identify critical topics, logical structure, illuminating examples, and the right level of detail. The classroom is your test-bed, an implacable mirror. Some of the kids are truly intrigued; this is what encourages you. Others are disgusted; this is what makes you try harder. Over the years, you figure out what is easy and what is difficult; what inspires and what bores; what deserves elaboration and what better be headed for a trash can.

Doctoral students are indispensable, too. How would I do without the innumerable discussions throughout those many years of graduate advising? Together we pored over literature surveys, formulated research questions, fine-tuned experiments, triumphed over successes, and lamented our failures. First they learned from me; then I learned from them.

Grateful thanks to all my students, undergraduate and graduate alike!

A scholar's mind is a product of countless encounters. Colleagues, mentors, bosses, scientists, friends, and rivals, how many of them have I met and talked to and argued with! Each has left a mark, even if a tiny one; they have made me who I am. This book would be a thick volume if I were to give credit to every single name, so let me narrow the field to those who offered a helping hand during the formative postdoctoral years, when my career still seemed a distant dream: Robert Trappl, Gerhard Widmer, Ivan Bratko, Stan Matwin, Rob Holte, Ryszard Michalski, Tom Mitchell, Pat Langley, Jan Zitkow, ... I have to stop, or the list would never end.

Grateful thanks to them all, thanks for all those memories!

Contents

Introduction to Third Edition

Only six years have passed, and so much has happened. And yet it seems only yesterday that the first edition of this textbook appeared! Its Introduction then began with the following words.

> Machine learning has come of age. And just in case you might think this an empty platitude, let me clarify.
>
> The dream that machines would one day be able to learn is as old as computers themselves, perhaps older still. For a long time, however, it remained just that: a dream. True, Rosenblatt's perceptron did trigger a wave of activity, but in retrospect, the excitement was short-lived. As for the attempts that followed, these fared even worse. Barely noticed, happily ignored, they never really made it: no start-up companies, no follow-up research, hardly any support from funding agencies. Machine learning remained an underdog, condemned to the miserable life in the shadows of luckier disciplines. The grand ambition lay dormant.
>
> And then it all changed.
>
> A group of visionaries pointed out a weak spot in the knowledge-based systems that were all the rage in the 1970s' artificial intelligence: where was the *knowledge* to come from? The prevailing wisdom of the day insisted that it should take the form of if-then rules put together by the joint effort of engineers and field experts. Practical experience, though, was unconvincing. Experts found it difficult to communicate what they knew to engineers. Engineers, in turn, were at a loss as to what questions to ask, and what to make of the answers. A few widely-publicized success stories notwithstanding, most attempts to create a knowledge base of, say, tens of thousands of such rules proved frustrating.
>
> The proposition made by the visionaries was as simple as it was audacious. If it is so hard to tell a machine exactly how to go about a certain problem, why not provide the instruction indirectly, conveying the necessary skills by way of examples from which the computer will—yes, learn!
>
> Of course, this only makes sense if we have the algorithms to do the learning—and this was the main difficulty. It turned out that neither Rosenblatt's perceptron, nor the techniques developed later were very useful. But the absence of the requisite machine-learning techniques was not an obstacle; rather, it presented a challenge that inspired quite a few brilliant minds. The idea of endowing computers with the ability to learn opened new horizons and gave rise to no small amount of excitement. The world was about to take notice.

The bombshell exploded in 1983. *Machine Learning: The AI Approach*[1] was a thick volume of research papers which introduced the most diverse ways of addressing the great mystery. Under their influence, a new scientific discipline was born—virtually overnight. Three years later, a follow-up book appeared, then another. A soon-to-become-prestigious scientific journal was founded. Annual conferences of undisputed repute were launched. And dozens, perhaps hundreds, of doctoral dissertations were submitted and successfully defended.

In this early stage, the question was not only *how* to learn, but also *what* to learn and *why*. In retrospect, those were wonderful times, so creative that the memories are tinged with nostalgia. It is only to be regretted that many great thoughts were later abandoned. Practical needs of realistic applications got the upper hand, pointing to the most promising avenues for further efforts. After a period of enchantment, concrete research strands crystallized: induction of if-then rules for knowledge-based systems; induction of classifiers, programs capable of improving their skills based on experience; automatic fine-tuning of Prolog programs; and others. So many were the directions that some leading personalities felt it necessary to try to steer further development by writing monographs, some successful, others less so.

A critical watershed was Tom Mitchell's legendary textbook.[2] This summarized the state-of-the-art of the field in a format appropriate for doctoral students and scientists alike. One by one, universities started offering graduate courses that were usually built around this book. Meanwhile, research methodology grew more systematic. A rich repository of machine-leaning test-beds was created, making it possible to compare the performance of diverse learning algorithms. Statistical methods of evaluation became wide-spread. Public-domain versions of most popular programs were made available. The number of scientists dealing with this discipline grew to thousands, perhaps even more.

Now we have reached the stage where a great many universities are offering Machine Learning as an undergraduate course. This is a new situation because classes of this kind call for a different sort of textbook. Apart from mastering the baseline techniques, the future engineers need to develop a good grasp of the strengths and weaknesses of alternative approaches, they have to become aware of the peculiarities and idiosyncrasies of competing paradigms. Above all, they must understand the circumstances under which some techniques succeed while others fail. Only then will they be able to make the right choices when dealing with concrete applications. A textbook that is to provide all of the above should contain less mathematics, but a lot of practical advice ...

So much for the 2015 Introduction.

Only six years, and so much water under the bridge! The world of machine-learning has changed, having advanced from being grudgingly accepted to dominating the mainstream. The field is now famous! Spectacular success stories are reported by TV channels, analyzed in newspapers, and passionately discussed in countless blogs. A generation ago, only dedicated specialists knew it even existed; today, every educated person has heard about computers beating world champions in chess, about language-understanding software, and about computer vision systems outperforming humans. The technology that makes computers learn from data and experience has become the magic wand that elevated Artificial Intelligence from hazy dreams to hard reality.

[1]Edited by R. Michalski, J. Carbonell, and T. Mitchell.

[2]T. Mitchell, *Machine Learning*, McGraw-Hill, 1997.

In view of all these staggering advancements, you would expect the author to rejoice: his passion has been vindicated, the object of his life-long studies now triumphs. There is a flip side, though. The elemental force of all those massive developments has rendered the first two editions of this book outdated. While the foundations are still as solid as ever, the center of gravity has shifted. What was hailed as revolutionary a few years ago, is now commonplace. What was labeled cutting-edge is now introductory. There is no escaping: the author is bound to sit down and start re-writing. Hence this third edition.

The most important topic that had to be added is *deep learning*; not because of being the talk of the town but by virtue of its demonstrated ability to tackle such daunting problems as learning to recognize complicated objects, scenes, and situations in digitized images, even in videos. An entire new chapter is now dedicated to convolutional neural networks, discussing the underlying principles, architectures, and practical use.

No less important is *reinforcement learning*. Whereas the book's previous editions only mentioned the related techniques in passing, it has now become necessary to devote to them no less than two full-length chapters. Some recent achievements are so spectacular that the textbook would seem ridiculously incomplete without adequate treatment of the relevant algorithms.

Unsupervised learning, too, has gained in importance, especially those mechanisms that from existing attributes create higher-level features to describe training examples. Also mechanisms capable of visualizing multidimensional data have become quite important. An entire new section dealing with *auto-encoding* had to be added. Most of the other sections of this chapter were expanded.

One new chapter now focuses on *temporal learning*. Even a beginner needs to know the basic principles of recurrent neural networks and needs to have an idea of what is *long short-term memory*. Another chapter introduces the reader into the realm of *Hidden Markov Models*. True, this is somewhat advanced, but the reader must at least know that it exists.

The older topics all remain, but in this new edition, the author sought to improve the clarity of exposition: to make the text easier to read and the pictures more pleasant to look at. These re-written chapters include the Bayesian rule, the nearest-neighbor principle, linear and polynomial classifiers, decision trees, artificial neural networks, and the boosting algorithms. Significant space is devoted to practical aspects of concrete engineering applications and to the ways of assessing their performance, including statistical evaluation.

In the course of his writing, the author kept reminding himself that his goal was an introductory text, not an encyclopedia. To prevent the volume from inflating beyond reasonable bounds, some hard decisions had to be made. What is it that really matters, what should better be eliminated? The answer is not immediately obvious. "Will the removal of X degrade the book's value?" "Is inclusion of Y really helpful?" These are the question that the author struggled with during those long months of work. In answering them, he relied on the following criteria.

1. Simplicity. Lofty theories are appreciated by scientists and seasoned professionals; for beginners, excessive sophistication is discouraging.
2. Programmer's perspective. Each algorithm should be easy to implement in a general-purpose programming language that does not rely on fancy built-in functions typical of highly specialized languages.
3. Consistency. All material should be of about the same level of difficulty, and it should rely on similar concepts and mathematical abstractions.
4. Personal preference. Whenever the previous three criteria made some alternatives appear interchangeable, the author followed his heart.

General philosophy is the same as in the previous editions: major chapters are divided into short sections such that each can be absorbed in one sitting. Each section is followed by 2–4 control questions to make sure the reader has not overlooked something serious. I find this important. Inadequate command of the fundamentals can make what follows hard to understand; such difficulty can cause antipathy, even a mental block, something to be avoided. To succeed, you need to enjoy what you're doing, and you will not enjoy it if you get lost early on.

Reading a textbook is not enough. One has to put in additional effort. To help navigate the reader in his or her further work, the author included at the end of each chapter a *Solidify Your Knowledge* section, usually consisting of three parts: exercises, thought experiments, and computer assignment. Each serves its own purpose, and there is no harm in making this explicit.

1. There is no learning without revision; and the best way to revise is by doing *exercises*. Those in this book are not supposed to be difficult, and there was no need for tricky catches.
2. Exercises are not enough. To master the technology, the engineer has to develop a "gut feeling" for the strengths and weaknesses of the individual techniques, to learn how to decide which of them to use under specific circumstances. One way to reach beyond mere introduction is to ask questions that seem to lack clear answers. To encourage this mental effort is the task for the *Give It Some Thought* questions.
3. The ultimate test of whether a certain algorithm has been mastered is a running computer program: implementation will force you to think through all the details that might otherwise go unnoticed. Practical experimentation with the developed software will help you appreciate subtleties that you may not have expected.

The reader will notice that the algorithms are presented in the form of pseudocode, and never in a concrete programming language. This is intentional. Programming languages come and go, their popularity waxes and wanes. An author who wants to see his book in print for a reasonable time to come does not want the text to depend on Python any more than on C++ or Fortran.

Miami, FL, USA
June 2021

Miroslav Kubat

Chapter 1
Ambitions and Goals of Machine Learning

You will find it difficult to describe your mother's face accurately enough for your friend to recognize her in a supermarket. But if you show him a few of her pictures, he will immediately see the tell-tale traits. As they say, a picture—an example—is worth a thousand words. Likewise, you will not become a professional juggler by just being told how to do it. The best any instructor can do is to offer some initial advice, and then let you practice and practice and practice—and learn from your own experience and failures.

This is what our technology is trying to emulate. Unable to define complicated concepts with adequate accuracy, we will convey them to the machine by way of examples. Unable to implement advanced skills, we instruct the computer to acquire them by systematic experimentation. Add to all this the ability to draw conclusions from the analysis of raw data, and to communicate them to the human user, and you know what the essence of *machine learning* is.

The task for this chapter is to elaborate on these ideas.

1.1 Training Sets and Classifiers

Let us first characterize the problem and introduce certain fundamental concepts that will keep us company us throughout the rest of the book.

Pre-Classified Training Examples Figure 1.1 shows six pies that Johnny likes, and six that he does not. In the sequel, we will refer to them as the *positive* and *negative examples* of the underlying concept. Together, they constitute a *training set* from which the machine is to induce a *classifier*—an algorithm capable of categorizing any future pie into one of the two *classes*: positive and negative.

The number of classes can of course be greater than just two. Thus a classifier that decides whether a landscape snapshot was taken in `spring`, `summer`, `fall`, or `winter` distinguishes four classes. Software that identifies characters scribbled

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_1

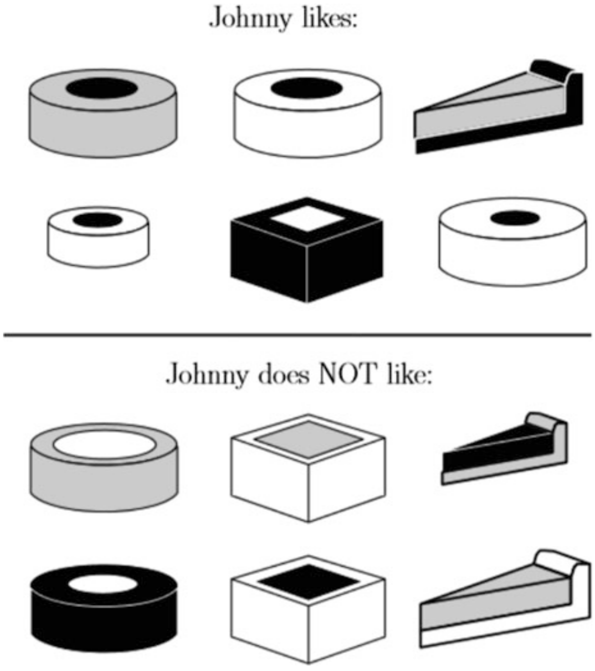


Fig. 1.1 A simple machine-learning task: induce a classifier capable of labeling future pies as positive and negative instances of “a pie that Johnny likes”

on a tablet needs at least 36 classes: 26 for letters and 10 for digits. And document-categorization systems are capable of identifying hundreds, even thousands of different topics. The only motivation for illustrating the input to machine learning by a two-class domain was its simplicity.

Attribute Vectors To be able to communicate the training examples to the machine, we have to describe them. The most common mechanism relies on the so-called *attributes*. In the “pies” domain, five may be suggested: `shape` (circle, triangle, and square), `crust-size` (thin or thick), `crust-shade` (white, gray, or dark), `filling-size` (thin or thick), and `filling-shade` (white, gray, or dark). Table 1.1 specifies the values of these attributes for the twelve examples in Fig. 1.1. For instance, the pie in the upper-left corner of the picture (the table calls it `ex1`) is described by the following conjunction:

```
(shape=circle) AND (crust-size=thick) AND (crust-shade=gray)
AND (filling-size=thick) AND (filling-shade=dark)
```

Classifier to Be Induced The training set constitutes the input from which we are to induce the classifier. But *what* classifier?

Suppose we want it in the form of a Boolean function that is *true* for positive examples and *false* for negative ones. Checking the expression `[(shape=circle) AND (filling-shade=dark)]` against the training set, we can see that it is *false* for all negative examples: while it *is* possible to find

Table 1.1 The twelve training examples described by attributes

		Crust		Filling		Class
Example	Shape	Size	Shade	Size	Shade	
ex1	Circle	Thick	Gray	Thick	Dark	`pos`
ex2	Circle	Thick	White	Thick	Dark	`pos`
ex3	Triangle	Thick	Dark	Thick	Gray	`pos`
ex4	Circle	Thin	White	Thin	Dark	`pos`
ex5	Square	Thick	Dark	Thin	White	`pos`
ex6	Circle	Thick	White	Thin	Dark	`pos`
ex7	Circle	Thick	Gray	Thick	White	`neg`
ex8	Square	Thick	White	Thick	Gray	`neg`
ex9	Triangle	Thin	Gray	Thin	Dark	`neg`
ex10	Circle	Thick	Dark	Thick	White	`neg`
ex11	Square	Thick	White	Thick	Dark	`neg`
ex12	Triangle	Thick	White	Thick	Gray	`neg`

negative examples that are circular, none of these has a dark filling. As for the positive examples, however, the expression is *true* for four of them and *false* for the remaining two. This means that the classifier makes two errors, a transgression we might refuse to tolerate, suspecting there is a better solution. Indeed, the reader will easily verify that the following expression never goes wrong on the entire training set:

```
[ (shape=circle) AND (filling-shade=dark) ] OR
[ NOT(shape=circle) AND (crust-shade=dark) ]
```

Problems with the Brute-Force Approach How does a machine find a classifier of this kind? Brute force (something that computers are so good at) will not do here. Just consider how many different examples can be distinguished by the given set of attributes in the "pies" domain. For each of the three different `shapes`, there are two alternative `crust-sizes`, the number of combinations being $3 \times 2 = 6$. For each of these, the next attribute, `crust-shade`, can acquire three different values, which brings the number of combinations to $3 \times 2 \times 3 = 18$. Extending this line of reasoning to *all* attributes, we realize that the size of the *instance space* is $3 \times 2 \times 3 \times 2 \times 3 = 108$ different examples.

Each subset of these examples—and there are 2^{108} subsets!—may constitute the list of positive examples of someone's notion of a "good pie." And each such subset can be characterized by at least one Boolean expression. Running each one of these classifiers through the training set is clearly out of the question.

Manual Approach and Search Uncertain about how to invent a classifier-inducing algorithm, we may try to glean some inspiration from an attempt to create a classifier "manually," by the good old-fashioned pencil-and-paper method. When doing so, we begin with some tentative initial version, say, `shape=circular`. Having checked it against the training set, we find it to be *true* for four positive examples, but also for two negative ones. Apparently, the classifier needs to be "narrowed" (specialized) so as to exclude the two negative examples. One way to go about the specialization is to add a conjunction, such as when turning `shape=circular` into `[(shape=circular) AND (filling-shade=dark)]`. This new expression, while *false* for all negative examples, is still imperfect because it covers only four (`ex1, ex2, ex4`, and `ex6`) of the six positive examples. The next step should therefore attempt some generalization, perhaps by adding a disjunction: { `[(shape=circular) AND (filling-shade=dark)] OR (crust-size=thick)` }. We continue in this way until we find a hundred-percent accurate classifier (if it exists).

The lesson from this little introspection is that the classifier can be created by means of a sequence of specialization and generalization steps which gradually modify a given version of the classifier until it satisfies certain predefined requirements. This is encouraging. Readers with background in Artificial Intelligence will recognize this procedure as a *search* through the space of Boolean expressions.

1.1.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- What is the input of the learning problem we have just introduced? What is its output?
- How do we describe the training examples? What is *instance space*? Can we calculate its size?
- In the "pies" domain, find a Boolean expression that correctly classifies all the training examples from Table 1.1.

1.2 Expected Benefits of the Induced Classifier

So far, we have measured the error rate by comparing the training examples' known classes with those recommended by the classifier. Practically speaking, though, our goal is *not* to reclassify objects whose classes we already know; what we really want is to label *future examples* of whose classes we are as yet ignorant. The classifier's anticipated performance on these is estimated experimentally. It is important to know how.

Independent Testing Examples The simplest scenario will divide the available pre-classified examples into two parts: the training set, from which the classifier is induced, and the *testing set*, on which it is evaluated (Fig. 1.2). Thus in the "pies" domain, with its 12 pre-classified examples, the induction may be carried out on randomly selected eight, and the testing on the remaining four. If the classifier then

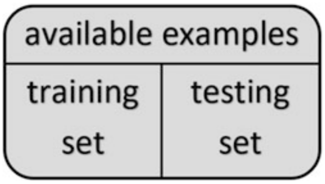


Fig. 1.2 Pre-classified examples are divided into the training and testing sets

"guesses" correctly the class of three testing examples (while going wrong on a single one), its performance is estimated as 75%.

Reasonable though this approach may appear, it suffers from a major drawback: a random choice of eight training examples may not be sufficiently representative of the underlying concept—and the same applies to the even smaller testing set. If we induce the meaning of a `mammal` from a training set consisting of a whale, a dolphin, and a platypus, the learner may be led to believe that mammals live in the sea (whale, dolphin), and sometimes lay eggs (platypus), hardly an opinion a biologist will endorse. And yet, another choice of training examples may result in a classifier satisfying the highest standards. The point is, a different training/testing set division gives rise to a different classifier—and also to a different estimate of future performance. This is particularly serious if the number of pre-classified examples is small.

Suppose we want to compare two machine-learning algorithms in terms of the quality of the products they induce. The problem of non-representative training sets can be mitigated by the so-called *random sub-sampling*.[1] The idea is to repeat the random division into the training and testing sets several times, always inducing a classifier from the i-th training set, and then measuring the error rate, E_i, on the i-th testing set. The algorithm that delivers classifiers with the lower average value of E_i's is deemed better—at least as far as classification performance is concerned.

Need for Explanations In some applications, establishing the class of each example is not enough. Just as desirable is to know the reasons behind the classification. Thus a patient is unlikely to give consent to amputation if the only argument in support of surgery is, "this is what our computer says." But how to find a better explanation?

In the "pies" domain, a lot can be gleaned from the Boolean expression itself. For instance, we may notice that a pie was labeled as negative whenever its shape was square, and its filling white. Combining this observation with other sources of knowledge may offer relevant insights: the dark shade of the filling may indicate poppy, an ingredient Johnny is known to love; or the crust of circular pies turns out to be more crispy than that of square ones; and so on. The knowledge obtained in this manner can be more desirable than the classification itself.

[1]Chapter 12 will describe some other methodologies.

Alternative Solutions Note that the given data make it possible to induce *many* classifiers that are perfect in the sense that they correctly label the entire training set. The "pies" domain contained 12 training examples, and the classes of the remaining 96 examples were unknown. Consider now only such classifiers that assign the correct label to each training example. One such classifier will label as positive the first of the remaining examples, and as negative all the other 95. Another will label as positive the first two of the remaining examples, and as negative the remaining 94 …and so on. The fact that for each of the 96 examples two possibilities exist (positive and negative) means that there are 2^{96} different ways in which some classifier can label them.

Shockingly, this means 2^{96} different classifiers can label correctly all training examples, while differing from each other in their behavior on the unknown 96. If we only look of a logical expression that correctly classifies the available pies, we may still get a classifier that will go wrong on any future example that has not been seen in the training set.

1.2.1 *What Have You Learned?*

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- How can we estimate the error rate on examples that have not been seen during learning? What is *random sub-sampling*?
- Why is error rate usually higher on the testing set than on the training set?
- Give an example of a domain where the classifier also has to explain its action, and an example of a domain where this is unnecessary.
- What do we mean by saying that, "there is a combinatorial number of classifiers that correctly classify all training examples"?

1.3 Problems with Available Data

The class recognition task, schematically represented by Fig. 1.3, is the most popular task of our discipline. Many concrete engineering problems can be cast in this framework: recognition of visual objects, understanding natural language, medical diagnosis, and identification of hidden patterns in scientific data. Each of these fields may rely on *classifiers* capable of labeling objects with the right classes based on the features, traits, and attributes characterizing these objects.

Origin of the Training Examples In some applications, the training set is created manually: an expert prepares the examples, tags them with class labels, chooses the attributes, and specifies the value of each attribute in each example. In other

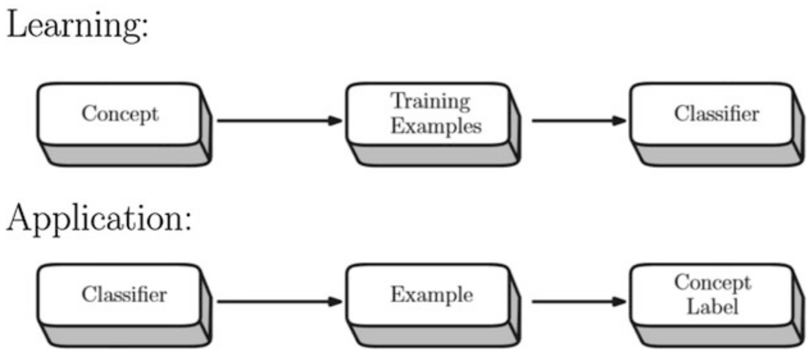


Fig. 1.3 Training examples are used to induce a classifier. The classifier is then used to classify future examples

domains, the process is computerized. For instance, a company may want to be able to anticipate an employee's intention to leave. Their database contains, for each person, the address, gender, marital status, function, salary raises, promotions—as well as the information about whether the person is still with the company or, if not, the day they left. From this, a program can obtain the attribute vectors, labeled as positive if the given person left within a year since the last update of the database record.

Sometimes, the attribute vectors are automatically extracted from a database and labeled by an expert. Alternatively, some examples can be obtained from a database and others added manually. Often, two or more databases are combined. The number of such variations is virtually unlimited.

But whatever the source of the examples, they are likely to suffer from imperfections whose essence and consequences the engineer has to understand.

Different Types of Attributes In the toy domain from Fig. 1.1, any of the attributes could only acquire one out of two or three different values. Such attributes are referred to as "discrete" or "discrete-valued." Other attributes, such as `age`, will be called "numeric" because their values are numbers, say, `age = 23`. Sometimes we want to emphasize that the numeric value is not necessarily an integer, coming as it does from a continuous domain, such as `price = 2.35`. In this case we will say that the attribute is "continuous-valued."

Irrelevant Attributes Some attributes are important, while others are not. While Johnny may be fond of poppy filling, his preference for a pie will hardly be driven by the cook's shoe size. This is something to be concerned about: *irrelevant* attributes add to computational costs; they can even mislead the learner. Can they be avoided?

Usually not. True, in manually created domains, the expert is supposed to know which attributes really matter, but even here, things are not so simple. Thus the author of the "pies" domain might have done her best to choose those attributes she believed to matter. But unsure about the real reasons behind Johnny's tastes, she may have included attributes whose necessity she suspected—but could not guarantee.

Even more often will the problems with relevance occur when the examples are extracted automatically from a database. Databases are developed primarily with the intention to provide access to lots of information—of which usually only a tiny part pertains to the learning task. As to which part this is, we usually have no idea.

Missing Attributes Conversely, some critical attributes can be missing. Mindful of his parents' finances, Johnny may be prejudiced against expensive pies. The absence of attribute `price` will then make it impossible to induce a good classifier: two examples, identical in terms of the available attributes, can differ in the values of the vital "missing" attribute. No wonder that, though identically described, one example is positive, and the other is negative. When this happens, we say that the training set is *inconsistent*. The situation is sometimes difficult to avoid: not only may the expert be ignorant of the relevance of attribute `price`; it may be impossible to provide this attribute's values, and the attribute thus cannot be used anyway.

Redundant Attributes Somewhat less damaging are attributes that are *redundant* in the sense that their values can be obtained from other attributes. If the database contains a patient's `date-of-birth` as well as `age`, the latter is unnecessary because its value can be calculated by subtracting `date-of-birth` from today's date. Fortunately, redundant attributes are less dangerous than irrelevant or missing ones.

Missing Attribute Values In some applications, the user may find it easy to identify the right set of attributes. Unfortunately, however, the values of some attributes may be unknown. For instance, a company's database may contain the information about the number of children only for some employees, and not for others. Or, in a database of a hospital's patients file, each patient has undergone only some laboratory tests. For these, the values are known; but since it is impossible (and unreasonable) to subject each patient to all available tests, most test results will be missing.

Attribute-Value Noise Attribute values and class labels often cannot be trusted on account of unreliable sources of information, poor measurement devices, typos, the user's confusion, and many other reasons. We say that the data suffer from various kinds of *noise*.

Stochastic noise is random. For instance, since our body-weight varies during the day, the reading we get in the morning is different from the one in the evening. A human error can also play a part: lacking the time to take a patient's blood pressure, a negligent nurse simply scribbles down a modification of the previous reading. By contrast, *systematic noise* drags all values in the same direction. For instance, a poorly calibrated thermometer always gives a lower reading than it should. Something else occurs in the case of *arbitrary artifacts*; here, the given value bears no relation to reality such as when an EEG electrode gets loose and, from that moment on, all subsequent readings will be zero.

Class-Label Noise Class labels suffer from similar problems as attributes. The labels recommended by an expert may not have been properly recorded; alternatively, some examples find themselves in a "gray area" between two classes, in which event the correct labels are not certain. Both cases represent stochastic noise, of which the latter will typically affect examples from the borderline region between the two classes. However, class-label noise can also be systematic: a physician may be reluctant to diagnose a rare disease unless the evidence is overwhelming—his class labels are then more likely to be negative than positive. Finally, arbitrary artefacts in class labels are encountered in domains where the classes are supplied by an automated process that has gone wrong.

Class-label noise is usually more dangerous than attribute-value noise. An incorrect value of an attribute may only slightly modify the example's overall characteristics and may thus only marginally affect the induced classifier. By contrast, a positive example mislabeled as negative can be highly misleading.

1.3.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Explain the meaning of the following terms: irrelevant and redundant attributes, missing attributes, and missing attribute values. Illustrate each of them using the "pies" domain.
- What is an "inconsistent training set"? What can be the cause? How can it affect the learning process?
- What kinds of noise do we know? What are their possible sources? In what way can noise affect the success and/or failure of the learning enterprise?

1.4 Many Roads to Concept Learning

The reader now understands that learning from pre-classified training examples is not easy. So many obstacles stand in the way. Even if the training set is perfect and noise-free, many classifiers can be found that are capable of correctly classifying all training examples but will differ in their treatment of examples that were not seen during learning. How to choose the best one?

Facing the Real World The training examples are rarely perfect. Most of the time, the class labels and attributes are noisy, a lot of the available information is irrelevant, redundant, or missing, the training set may be far too small to capture all critical aspects—the list goes on and on. There is no simple solution. No wonder that an entire scientific discipline—machine learning—has come to being that seeks to

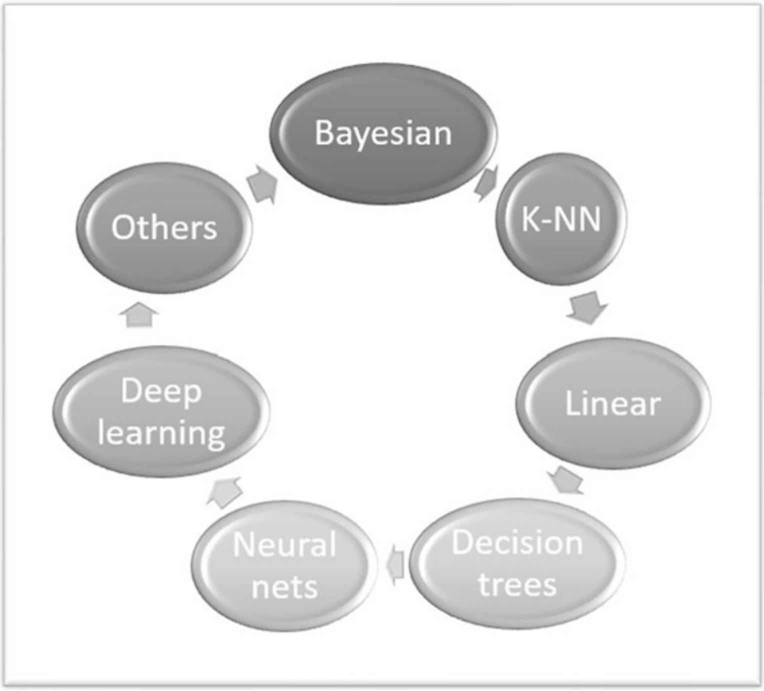


Fig. 1.4 For class recognition, quite a few competing paradigms have been developed, each with advantages and shortcomings

come to grips with all the above-mentioned issues and to illuminate all the tangled complications of the underlying tasks.

As pointed out by Fig. 1.4, engineers have at their disposal several major and some smaller paradigms, each marked by different properties, each exhibiting different strengths and shortcomings when applied to a concrete task. To show the nature of each of these frameworks, and to explain how it behaves under diverse circumstances is the topic for the rest of this book. But perhaps we can mention here at least some of the basic principles.

Probabilities We have seen that many classifiers can be induced that classify perfectly the training data—but each with a different behavior on future data. Which of the classifiers is the engineer to use? One way to avoid this difficulty is to rely on the time-tested theory of probability. The relative frequencies of circular or square pies in the training set surely give hints of a future pie being from the positive or negative class? This is the avenue followed by the so-called *Bayesian classifiers*.

Similarities Another idea is to rely on similarities. Surely we may expect that objects belonging to the same class have something in common that they are in some way similar? This reasoning is behind the so-called *nearest-neighbor* classifiers. The assumption is that mutual similarity of a pair of examples is captured by the geometric distance between the attribute vectors describing them.

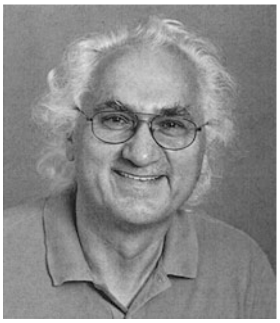


Fig. 1.5 Learning to recognize specific objects: how many eyes does the person have? is there a nose, in the picture? an airplane?

Decision Surfaces Another major philosophy is built around the multidimensional "space metaphor." Suppose, for simplicity, that all attributes are numeric so that each example can be identified with a single point in an N-dimensional space where N is the number of attributes. If it is true that examples of the same class tend to find themselves close to each other, geometrically (see the previous paragraph), then it should be possible to delineate a region occupied by positive examples and another region occupied by negative examples. These regions can then be separated by a "decision surface": positive examples on one side and negative examples on the other. The reader will see that this is a very potent idea; a few chapters focus on the various ways of implementing it, among them *linear classifiers, decision trees*, and *artificial neural networks*.

Advanced Issues Once we have acquainted ourselves with the fundamental building blocks, and once we have learned to appreciate their subtle points, we are ready to probe further. For example, classification performance can be increased by combining groups of classifiers that vote. The behavior of baseline classifiers is often compromised by circumstances known under fancy names as concept drift, imbalanced classes, and bias. These will be discussed at great length and so will be mathematical formulas quantifying the very notion of learnability.

Deep Learning Some of the currently most famous breakthroughs in machine learning were achieved by novel mechanisms that convert low-level attributes (e.g., each giving the intensity of one pixel on a computer screen) into meaningful high-level features which are then used for recognition purposes. Since these conversion mechanisms usually employ artificial neural networks with several (or many) layers, the tools came to be known under the collective name of *deep learning*.

This relatively new technology has become famous thanks to well-publicized breakthroughs in computer vision. For instance, even undergraduate courses now teach how to write computer program that learns to recognize in a picture (such as the one in Fig. 1.5) certain specific objects such as eyes or noses.

Understanding the Principles Machine learning is more art than science. Instead of relying on ready-made recipes, the engineer has to understand the critical aspects of the task at hand; these aspects then guide his or her decisions. Figure 1.6 reminds

Fig. 1.6 Alternative machine-learning paradigms offer different tools. The engineer needs to know how to choose the best tool for the given task

us that the paradigms briefly summarized in this section are nothing but tools from which the engineer chooses the one best-suited for the task at hand. It stands to reason that such choice is only possible if the idiosyncrasies of each tool are properly understood, and the peculiarities of the concrete application appreciated.

1.5 Other Ambitions of Machine Learning

Induction of classifiers is the most popular machine-learning task—but not the only one! Let us briefly survey some of the other topics covered in this book.

Unsupervised Learning A lot of information can be gleaned even from examples that are not labeled with classes. To begin with, analysis can reveal that the examples create clusters of similar attribute vectors. Each such cluster can exhibit different properties that may deserve to be studied.

We also know how to map unlabeled N-dimensional vectors to a neural field. The resulting two-dimensional matrix helps visualize the data in ways different from classical cluster analysis. One can see which parts of the instance space are densely populated and which parts sparsely, we may even learn how many exceptions there are. Approaches based on the so-called auto-encoding can create from existing attributes meaningful higher-level attributes; such re-description often facilitates learning in domains marked by excessive detail.

Reinforcement Learning Among the major triumphs of machine learning, perhaps the most fascinating are computers beating the best humans in such games as chess, Backgammon, and Go. For generations, such feats were deemed impossible! And yet, here we are. Computer programs can learn to become proficient simply by playing innumerable games against themselves—and by learning from this experience. What other proof of the potential of our discipline does anybody want?

The secret behind these accomplishments is the techniques known as *reinforcement learning*, frequently in combination with artificial neural networks and deep learning. The application field is much broader than just game playing. The idea is to let the machine develop an ability to act in real-world environments, to react to changes in this environment, to optimize its behavior in tasks ranging from pole-balancing to vehicle navigation to advanced decision-making in domains that lack detailed technical description.

Hidden Markov Models Can we estimate how many years in the fourteenth century were hot, and how many were cold, if the only information at our disposal are tree rings? The answer is yes—provided that we know the probabilities that in the hot or cold years the tree rings are small, medium, or large; and if we know how likely it is, for instance, that a cold year is followed by a hot year. Lacking direct measurements of temperatures in the middle ages, we can still develop reasonable opinions from the indirect information provided by the tree rings.

The issue just described characterizes a whole class of tasks where the goal is to make time-series predictions based on indirect variables. Problems of this sort have been applied to an impressive range of applications, including finance, natural-language processing, bioinformatics, and finance.

The task for machine learning is to induce from available data a reliable model consisting of various probabilities: how likely is it that state X will be followed by state Y, how likely it is that observation A is made if the underlying state is X, and so on.

Odds and Pieces The author wanted to make this book as useful to engineers as possible. This is why the text includes simple but instructive applications that highlight practical issues often neglected by more theoretical volumes. Special attention has been paid to performance evaluation, an issue that is trickier than a beginner suspect, calling for a whole range of diverse metrics and experimental methodologies. One chapter explains classical methods of statistical evaluation.

1.6 Summary and Historical Remarks

- Induction from a training set of pre-classified examples is the most deeply studied machine-learning task.
- Historically, the task is cast as search. This, however, is not enough. The book explores a whole range of more useful techniques.
- Classification performance is estimated with the help of pre-classified testing data. The simplest performance criterion is error rate, the percentage of examples misclassified by the classifier.
- Two classifiers that both correctly classify all training examples may differ significantly in their handling of future examples.
- Apart from low error rate, some applications require that the classifier provides the reasons behind the classification.
- The quality of the induced classifier depends on training examples. The quality of the training examples depends not only on their choice but also on the attributes used to describe them. Some attributes are relevant, others irrelevant or redundant. Quite often, critical attributes are missing.
- The attribute values and class labels may suffer from stochastic noise, systematic noise, and random artefacts. The value of an attribute in a concrete example may not be known.

Historical Remarks In the 1998s and 1990s the machine-learning task was usually understood as an AI search. While several “founding fathers” came to see things this way independently of each other, Mitchell (1982) is often credited with being the first to promote the search-based approach; just as influential, however, was the family of AQ-algorithms proposed by Michalski (1969). The discipline got a major boost by the collection of papers edited by Michalski et al. (1983), a book that framed the mindset of a whole generation of scientists. The interested reader will find more about search techniques in textbooks of Artificial Intelligence, of which perhaps the most comprehensive is Russell and Norvig (2003).

The reader may find it interesting that the question of proper representation of concepts or classes intrigued philosophers for centuries. Thus John Stuart Mill (1865) explored concepts that are close to what the next chapter calls *probabilistic* representation; and William Whewel (1858) advocated *prototypical* representations that remind us of the subject of our Chap. 3.

The example (my favorite) of tree rings helping us to decide which years were hot or cold, in the past, is taken from Stamp (2018).

1.7 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter’s ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

1.7.1 Exercises

1. What is the size of the instance space in a domain where examples are described by ten Boolean attributes? How large is then the space of classifiers?

1.7.2 Give It Some Thought

1. In the “pies” domain, the size of the space of all classifiers is 2^{108}, provided that any subset of the instance space can represent a different classifier. How much will the search space shrink if we permit only classifiers in the form of conjunctions of attribute-value pairs?
2. What kind of noise do you think is possible in the “pies” domain? What can be the source of this noise? What other issues may render training sets of this kind less than perfect?

3. Some classifiers behave as black boxes that do not offer much in the way of explanations. Suggest examples of domains where black-box classifiers are impractical, and suggest domains where this limitation does not matter.
4. Consider the data-related difficulties summarized in Sect. 1.3. Which of them are really serious, and which can be tolerated?
5. What is the difference between redundant attributes and irrelevant attributes?
6. Take a class that you think is difficult to describe—for instance, the recognition of a complex biological object (oak tree, ostrich, etc.) or the recognition of a music genre (rock, folk, jazz, etc.). Suggest the list of attributes to describe potential training examples. Will the values of these attributes be easy to obtain? Which of the problems discussed in this chapter do you expect will complicate the learning process?
7. One strand of machine-learning research focuses on learning from examples that are not labeled with classes. What do you think can be the practical benefits of such programs?
8. The successes of *reinforcement learning* in game playing are impressive, but perhaps not very useful in the real world, say, in industry or economy. Can you think of some more practical application domain that might benefit from these techniques?

1.7.3 Computer Assignments

1. Write a program that will implement the search for the description of the "pies that Johnny likes." Define your own generalization and specialization operators. The evaluation function will rely on the error rate observed on the training examples.

Chapter 2
Probabilities: Bayesian Classifiers

The earliest attempts to predict an example's class from the knowledge of its attribute values go back to well before World War II—prehistory, by the standards of computer science. Of course, nobody used the term "machine learning," in those days, but the goal was essentially the same.

The strategy was to calculate for the given object (attribute vector) the probabilities of its belonging to the individual classes, and identified the class with the highest probability. This chapter will explain how to do this in the case of discrete attributes, and then in the case of numeric attributes.

2.1 The Single-Attribute Case

Let us start with something so simple as to be almost unrealistic: a domain where each example is described with a single attribute. Once we have grasped the principles of Bayesian classifiers under these simplified circumstances, we will generalize the idea for more realistic settings.

Prior probability and conditional probability. Let us return to the toy domain from the previous chapter. The training set consists of twelve pies ($N_{all} = 12$), of which six are positive examples of the given class ($N_{pos} = 6$) and six are negative ($N_{neg} = 6$). Assuming that the examples represent faithfully the given domain, the probability of Johnny liking a randomly picked pie is fifty percent because fifty percent of the training examples are positive.

$$P(\mathtt{pos}) = \frac{N_{pos}}{N_{all}} = \frac{6}{12} = 0.5 \tag{2.1}$$

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_2

Let us now choose one of the attributes, say, `filling-size`. The training set contains eight examples with thick filling ($N_{thick} = 8$), of which three are labeled as positive ($N_{pos|thick} = 3$). We say that the *conditional probability* of an example being positive given that `filling-size=thick` is 37.5%: the relative frequency of positive examples among those with thick filling indicates:

$$P(\texttt{pos}|\texttt{thick}) = \frac{N_{pos|thick}}{N_{thick}} = \frac{3}{8} = 0.375 \tag{2.2}$$

Conditional Probability and Classification The relative frequency of the positive class was calculated only for pies characterized by the given attribute value. Among the same eight pies with thick filling, five belonged to the negative class, $P(\texttt{neg}|\texttt{thick}) = 5/8 = 0.625$. Observing that $P(\texttt{neg}|\texttt{thick}) > P(\texttt{pos}|\texttt{thick})$, we conclude that the probability of Johnny disliking a pie with thick filling is greater than the probability of his liking it. For this reason, a classifier based on the probabilistic principle will label any examples with `filling-size=thick` as a negative instance of the "pie that Johnny likes."

Conditional probability, $P(\texttt{pos}|\texttt{thick})$, inspires more confidence than the prior probability, $P(\texttt{pos})$, because it uses more information. In a DayCare center with about as many boys as girls, we expect a randomly selected child to be a boy with probability $P(\texttt{boy}) = 0.5$. But the moment we hear someone refer to the child as Johnny, we increase this expectation because we know that a girl is less likely to be called by this name: $P(\texttt{boy}|\texttt{Johnny}) > P(\texttt{boy})$.

Joint Probability Conditional probability must not be confused with *joint probability* of two events occurring simultaneously. Be sure to use the right notation: in joint probability, the terms are separated by a comma, $P(\texttt{pos}, \texttt{thick})$; in conditional probability, by a vertical bar, $P(\texttt{pos}|\texttt{thick})$. Whereas $P(\texttt{pos}, \texttt{thick})$ denotes the probability that the example is positive *and* its filling is `thick`, $P(\texttt{pos}|\texttt{thick})$ is the probability of a positive example among those with `filling-size=thick`.

Numeric Example Figure 2.1 illustrates the terms just introduced. The rectangle represents all pies. The positive examples are in one circle and those with `filling-size=thick` in the other. The three instances in the intersection of the two circles satisfy both conditions. Finally, one pie satisfies neither, which is why it finds itself outside both circles. The conditional probability $P(\texttt{pos}|\texttt{thick}) = 3/8$ is obtained by dividing the size of the intersection (three) by the size of the circle `thick` (eight). The joint probability, $P(\texttt{pos}, \texttt{thick}) = 3/12$, is obtained by dividing the size of the intersection (three) by the size of the entire training set (twelve). The prior probability of $P(\texttt{pos}) = 6/12$ is obtained by dividing the size of the circle `pos` (six) with the size of the entire training set (twelve).

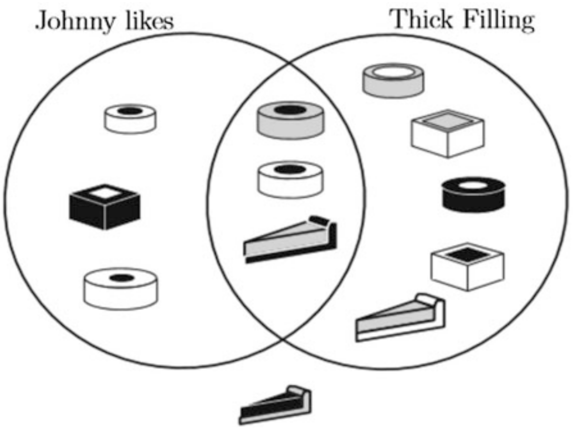


Fig. 2.1 The key terms: prior probabilities, $P(\texttt{pos}) = \frac{6}{12}$ and $P(\texttt{thick}) = \frac{8}{12}$; conditional probabilities, $P(\texttt{pos}|\texttt{thick}) = \frac{3}{8}$ and $P(\texttt{thick}|\texttt{pos}) = \frac{3}{6}$; and the joint probability, $P(\texttt{likes}, \texttt{thick}) = \frac{3}{12}$

Obtaining Conditional Probability from Joint Probability Figure 2.1 convinces us that joint probability can be calculated from prior probability and conditional probability:

$$P(\texttt{pos}, \texttt{thick}) = P(\texttt{pos}|\texttt{thick}) \cdot P(\texttt{thick}) = \frac{3}{8} \cdot \frac{8}{12} = \frac{3}{12}$$

$$P(\texttt{thick}, \texttt{pos}) = P(\texttt{thick}|\texttt{pos}) \cdot P(\texttt{pos}) = \frac{3}{6} \cdot \frac{6}{12} = \frac{3}{12}$$

Note that joint probability can never exceed the value of the corresponding conditional probability, $P(\texttt{pos}, \texttt{thick}) \leq P(\texttt{pos}|\texttt{thick})$, because conditional probability is multiplied by prior probability, $P(\texttt{thick})$ or $P(\texttt{pos})$, and this can never be greater than 1.

Notice, further, that $P(\texttt{thick}, \texttt{pos}) = P(\texttt{pos}, \texttt{thick})$ because both represent the probability of thick filling and positive class occurring in the same example. The left-hand sides of the previous two formulas thus being equal, the right-hand sides have to equal, too:

$$P(\texttt{pos}|\texttt{thick}) \cdot P(\texttt{thick}) = P(\texttt{thick}|\texttt{pos}) \cdot P(\texttt{pos})$$

Dividing both sides by $P(\texttt{thick})$, we obtain the Bayes formula that will provide the foundations for the rest of this chapter:

$$P(\texttt{pos}|\texttt{thick}) = \frac{P(\texttt{thick}|\texttt{pos}) \cdot P(\texttt{pos})}{P(\texttt{thick})} \tag{2.3}$$

Similar reasoning results in the formula for the opposite case, the probability that a pie with `filling-size = thick` belongs to the negative class.

$$P(\texttt{neg}|\texttt{thick}) = \frac{P(\texttt{thick}|\texttt{neg}) \cdot P(\texttt{neg})}{P(\texttt{thick})} \tag{2.4}$$

Comparing the values obtained from these two formulas, we decide which of the two classes, pos of neg, is more likely to be correct.

Practical calculations are even simpler than that. Seeing that the denominator, $P(\texttt{thick})$, is the same for both classes, we realize that we can just as well ignore it and just choose the class with the higher numerator.

Numeric Example The use of the Bayesian formula is illustrated by Table 2.1 that, for the sake of simplicity, deals with the trivial case where the examples are described by a single Boolean attribute. So simple is the single-attribute world that we might have obtained $P(\texttt{pos}|\texttt{thick})$ and $P(\texttt{neg}|\texttt{thick})$ directly from the training set, without having to resort to the Bayes formula. This is how we can verify the correctness of the results.

When the examples are described by two or more attributes, the probabilities are calculated in much the same way. Usually, however, the simplifying assumption of

Table 2.1 Illustrating the principle of Bayesian classification

Let the training examples be described by a single attribute, filling-size, whose value is either thick or thin. We want the machine to recognize the positive class (pos). Below are eight training examples:

	ex1	ex2	ex3	ex4	ex5	ex6	ex7	ex8
Size	thick	thick	thin	thin	thin	thick	thick	thick
Class	pos	pos	pos	pos	neg	neg	neg	neg

The probabilities of the individual attribute values and of the class labels are obtained by relative frequencies. For instance, three out of the eight examples are characterized by filling-size=thin; therefore, $P(\texttt{thin}) = 3/8$.

$P(\texttt{thin}) = 3/8$

$P(\texttt{thick}) = 5/8$

$P(\texttt{pos}) = 4/8$

$P(\texttt{neg}) = 4/8$

The conditional probability of a concrete attribute value within a given class is, again, determined by relative frequency. Our training set yields the following values:

$P(\texttt{thin}|\texttt{pos}) = 2/4$

$P(\texttt{thick}|\texttt{pos}) = 2/4$

$P(\texttt{thin}|\texttt{neg}) = 1/4$

$P(\texttt{thick}|\texttt{neg}) = 3/4$

Employing these values in the Bayes formula, we obtain the following conditional probabilities:

$P(\texttt{pos}|\texttt{thin}) = 2/3$

$P(\texttt{pos}|\texttt{thick}) = 2/5$

$P(\texttt{neg}|\texttt{thin}) = 1/3$

$P(\texttt{neg}|\texttt{thick}) = 3/5$

(note that $P(\texttt{pos}|\texttt{thin}) + P(\texttt{neg}|\texttt{thin}) = P(\texttt{pos}|\texttt{thick}) + P(\texttt{neg}|\texttt{thick}) = 1$)

Seeing that $P(\texttt{pos}|\texttt{thin}) > P(\texttt{neg}|\texttt{thin})$, we conclude that an example with filling-size=thin should be classified as positive. An example with filling-size = thick is deemed negative because $P(\texttt{neg}|\texttt{thick}) > P(\texttt{pos}|\texttt{thick})$.

mutual independence of attributes is used. The concrete mechanism of doing so will be presented in the next section.

2.1.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- How is the Bayes formula derived from the relation between conditional probability and joint probability?
- What makes the Bayes formula so useful? What does it enable us to calculate?
- Can the joint probability, $P(x, y)$, have a greater value than the conditional probability, $P(x|y)$? Under what circumstances is $P(x|y) = P(x, y)$?

2.2 Vectors of Discrete Attributes

Let us now proceed to a simple way of using the Bayes formula in more realistic domains where the examples are described by vectors of attributes such as $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, and where there are more than two classes.

Multiple Classes Many realistic applications are marked by more than two classes, not just the pos and neg from the "pies" domain. If c_i is the label of the i-th class, and if $\mathbf{x}$ is the vector describing the object we want to classify, the Bayes formula acquires the following form:

$$P(c_i|\mathbf{x}) = \frac{P(\mathbf{x}|c_i)P(c_i)}{P(\mathbf{x})}$$

The denominator being the same for each class, we choose the class that maximizes the numerator, $P(\mathbf{x}|c_i)P(c_i)$. Here, $P(c_i)$ is estimated by the relative frequency of c_i in the training set. With $P(\mathbf{x}|c_i)$, however, things are not so simple.

A Vector's Probability $P(\mathbf{x}|c_i)$ is the probability that a randomly selected instance of class c_i is described by vector $\mathbf{x}$. Can the value of this probability be estimated by relative frequency? Not really. In the "pies" domain, the size of the instance space was 108 different examples, of which the training set contained twelve, while none of the other vectors (the vast majority!) was represented at all. Relative frequency would indicate that the probability of $\mathbf{x}$ being positive is $P(\mathbf{x}|\text{pos}) = 1/6$ if we find $\mathbf{x}$ among the positive training examples, and $P(\mathbf{x}|\text{pos}) = 0$ if we do not. In other words, any $\mathbf{x}$ identical to some training example "inherits" this example's class label; if the vector is *not* in the training set, we have $P(\mathbf{x}|c_i) = 0$ for any c_i. In this case, the numerator in the Bayes formula will always be $P(\mathbf{x}|c_i)P(c_i) = 0$, which makes it impossible for us to choose the most probable class. Evidently, we are not getting very far trying to calculate the probability of an event that occurs only once—if it occurs at all.

The situation improves if only individual attributes are considered. For instance, `shape=circle` occurs four times among the positive examples and twice among the negative, the corresponding probabilities thus being $P(\texttt{shape} = \texttt{circle}|\texttt{pos}) = 4/6$ and $P(\texttt{shape} = \texttt{circle}|\texttt{neg}) = 2/6$. We see that, if an attribute can acquire only two or three values, chances are high that each of these values is represented in the training set more than once, thus providing better grounds for probability estimates.

Mutually Independent Attributes We need a formula that takes the probabilities of individual attribute values and uses them to estimate the probability that a vector of these values is found in a specific class, $P(\mathbf{x}|c_i)$. As long as the attributes are independent of each other, this is simple. If $P(x_i|c_j)$ is the probability that the value of the i-th attribute of an example in class c_j is x_i, then the probability, $P(\mathbf{x}|c_j)$, that a random representative of c_j is described by $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, is calculated as follows:

$$P(\mathbf{x}|c_j) = \prod_{i=1}^{n} P(x_i|c_j) \tag{2.5}$$

A mathematician will tell us that this is the probability of mutually independent events occurring simultaneously.

In the context of Bayesian classifiers this means that $\mathbf{x}$ should be labeled with class c_j if this class maximizes the following version of the Bayes formula's numerator:

$$P(c_j) \cdot \prod_{i=1}^{n} P(x_i|c_j) \tag{2.6}$$

Naive Bayes Assumption The assumption of mutually independent attributes is rarely justified. Indeed, can the interrelation of diverse variables ever be avoided? An object's weight grows with its size, the quality of health care may be traced to living standards, a plant's color can be derived from specific chemical properties. In short, domains where no two attributes are in any way dependent on each other are hard to find. No wonder that the above-described approach has acquired the unflattering name of *Naive Bayes*.

Yet practical experience is rather good—and perhaps this should not come as a surprise. True, the violation of the "independence requirement" renders all probability estimates somewhat suspect. But poor probability estimates may not necessarily lead to incorrect classification. We said that $\mathbf{x}$ should be labeled with the class that maximizes $P(\mathbf{x}|c_i) \cdot P(c_i)$. If the probability estimate is 0.8 for one class and 0.2 for the other, then the classifier's behavior will be unaffected even if the true values are, say, [0.6, 0.4] instead of the estimated [0.8, 0.2]. While expecting that the attributes should in principle be mutually independent, nothing much will happen if in reality they are not.

When Mutual Independence Cannot Be Assumed This said, we need to know how to handle a situation where the mutual interdependence of attributes is so strong that it *cannot* be ignored. A scientist's first instinct may be to resort to more advanced ways of estimating $P(\mathbf{x}|c_i)$. Unfortunately, while such methods do exist, their complexity becomes unmanageable with the growing number of attributes, and they tend to rely on terms whose values are hard to establish.

A more pragmatic approach will therefore attempt to reduce the attribute interdependence by data pre-processing. A good way to start is to eliminate redundant attributes, those whose values can be obtained from others. For instance, if the set of attributes contains `age, date-of-birth`, and `current-date`, chances are that Naive Bayes will do well if we use only one of them.

We can also try to replace two or three attributes by an artificially created one that combines them. Thus in the "pies" domain, a baker might have told us that `filling-size` is not independent of `crust-size`: if one is `thick`, he prefers to make the other `thin` and vice versa. In this event, we will benefit from replacing the two attributes with a new one, say, `CF-size`, which acquires only two values: `thick-crust-and-thin-filling` or `thin-crust-and-thick-filling`.

In the last resort, if we are uncomfortable with advanced methods of multivariate probability estimates, and if we want to avoid data pre-processing, these are always the possibility of giving up on Bayesian classifiers altogether, preferring other machine-learning paradigms.

Numeric Example To get used to the mechanisms that use Naive Bayes for classification, the reader is advised to go through the example detailed in Table 2.2. Here the class of a previously unseen pie is established with the help of the training set from Table 1.1, and with the help of the assumption that all attributes are mutually independent.

For the reader's convenience, Table 2.3 contains a pseudo code summarizing the procedure just explained.

2.2.1 *What Have You Learned?*

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Why do we want to assume that the attributes are mutually independent? What benefit does this assumption bring us?
- Why is this assumption often harmless even in domains where the attributes are *not* mutually independent?
- What can be done in domains where the mutual independence of attributes cannot be ignored?

Table 2.2 Bayesian classification in a domain where examples are described by vectors of independent attributes

Suppose we want to apply the Bayesian formula to the training set from Table 1.1 in order to determine the class of the following object:

```
x = [shape=square, crust-size=thick, crust-shade=gray
      filling-size=thin, filling-shade=white]
```

There are two classes, `pos` and `neg`. The procedure is to calculate the numerator of the Bayes formula separately for each class, and then choose class where the value is higher. In the training set, each class has the same number of representatives: $P(\texttt{pos}) = P(\texttt{neg}) = 0.5$. The remaining terms, $\prod_{i=1}^{n} P(x_i|\texttt{pos})$ and $\prod_{i=1}^{n} P(x_i|\texttt{neg})$, are calculated from the following conditional probabilities:

P(shape=square | pos)	= 1/6	P(shape=square | neg)	= 2/6
P(crust-size=thick | pos)	= 5/6	P(crust-size=thick | neg)	= 5/6
P(crust-shade=gray | pos)	= 1/6	P(crust-shade=gray | neg)	= 2/6
P(filling-size=thin | pos)	= 3/6	P(filling-size=thin | neg)	= 1/6
P(filling-shade=white | pos)	= 1/6	P(filling-shade=white | neg)	= 2/6

From these numbers, we obtain the following conditional probabilities:

$$P(\mathrm{x}|\texttt{pos}) = \prod_{i=1}^{n} P(x_i|\texttt{pos}) = \frac{1}{6}\cdot\frac{5}{6}\cdot\frac{1}{6}\cdot\frac{3}{6}\cdot\frac{1}{6} = \frac{15}{6^5}$$

$$P(\mathrm{x}|\texttt{neg}) = \prod_{i=1}^{n} P(x_i|\texttt{neg}) = \frac{2}{6}\cdot\frac{5}{6}\cdot\frac{2}{6}\cdot\frac{1}{6}\cdot\frac{2}{6} = \frac{40}{6^5}$$

Since $P(\texttt{pos}) = P(\texttt{neg}) = 0.5$, we see that $P(\mathbf{x}|\texttt{pos})\cdot P(\mathbf{pos}) < P(\mathbf{x}|\texttt{neg})\cdot P(\mathbf{neg})$. Therefore, we label **x** with the negative class

Table 2.3 Classification with Naive Bayes

The example to be classified is described by $\mathbf{x} = (x_1, \dots, x_n)$.

1. For each x_i, and for each class c_j, calculate the conditional probability, $P(x_i|c_j)$, as the relative frequency of x_i among all training examples from c_j.
2. For each class, c_j, carry out the following two steps:
   (i) estimate $P(c_j)$ as the relative frequency of this class in the training set;
   (ii) calculate the conditional probability, $P(\mathbf{x}|c_j)$, using the "naive" assumption of mutually independent attributes:
   $$P(\mathbf{x}|c_j) = \prod_{i=1}^{n} P(x_i|c_j)$$
3. Choose the class with the highest value of the product $P(c_j)\prod_{i=1}^{n} P(x_i|c_j)$.

2.3 Rare Events: An Expert's Intuition

For simplicity, probability is often estimated by relative frequency. Having observed phenomenon x thirty times in one hundred trials, we conclude that its probability is $P(x) = 0.3$. This is how we did it in the previous sections.

Estimates of this kind, however, can be trusted only when based on a great many observations. While it is conceivable that a coin flipped four times comes up heads three times, it would be silly to jump to the conclusion that $P(\texttt{heads}) = 0.75$. The physical nature of the experiment suggests otherwise: a fair coin should come up heads fifty percent of the time. Can this *prior expectation* help us improve our probability estimates in domains with few observations?

The answer is yes. Prior expectations are employed in the so-called m-estimates.

Essence of m-Estimates Let us consider experiments with a coin that may be fair—or unfair. In the absence of any extra information, our estimate of the probability of *heads* will be $\pi_{heads} = 0.5$. But how confident are we in this estimate? This is quantified by an auxiliary parameter, m, that informs the class-predicting program about the amount of our uncertainty. The higher the value of m, the more the probability estimate, $\pi_{head} = 0.5$, is to be trusted.

Returning to our experimental coin-flipping, let us denote by N_{all} the total number of trials, and by N_{heads} the number of "heads" observed in these trials. The following formula combines these values with the prior estimate and with our confidence in this estimate's reliability:

$$P_{heads} = \frac{N_{heads} + m\pi_{heads}}{N_{all} + m} \tag{2.7}$$

Note that the formula degenerates to the prior estimate if no experimental evidence has yet been accumulated, in which case, $P_{heads} = \pi_{heads}$ because $N_{all} = N_{heads} = 0$. Conversely, the formula converges to that of relative frequency if N_{all} and N_{heads} are so big as to render negligible the terms $m\pi_{heads}$ and m.

With $\pi_{heads} = 0.5$ and $m = 2$, we obtain the following:

$$P_{heads} = \frac{N_{heads} + 2 \times 0.5}{N_{all} + 2} = \frac{N_{heads} + 1}{N_{all} + 2}$$

Numeric Example Table 2.4 shows how the values thus calculated gradually evolve, step by step, in the course of five coin flips. The reader can see that m-estimate is for small numbers of experiments more in line with common sense than is relative frequency. Thus after two trials, m-estimate suggests a 0.75 chance of *heads*, whereas relative frequency interprets the two experiments as indicating a zero chance of *tails*, a statement that is hard to accept. Only as the number of trials increases do the values offered by m-estimate and relative frequency converge.

Impact of the User's Confidence Let us take a closer look at the effect of m, the parameter that quantifies our confidence in our prior estimate. A lot is revealed if

Table 2.4 For each of the five successive trials, the second row gives the observed outcome; the third, the relative frequency of *heads*; the last, the m-estimate of the *head*'s probability under the assumptions $\pi_{heads} = 0.5$ and $m = 2$

Trial number	1	2	3	4	5
Outcome	Heads	Heads	Tails	Heads	Tails
Relative frequency	1.00	1.00	0.67	0.75	0.60
m-estimate	0.67	0.75	0.60	0.67	0.57

we compare the two different alternatives calculated below: $m = 100$ on the left and $m = 1$ on the right (in both cases, $\pi_{heads} = 0.5$).

$$\frac{N_{heads} + 50}{N_{all} + 100} \qquad\qquad \frac{N_{heads} + 0.5}{N_{all} + 1}$$

The version with $m = 100$ allows the prior estimate to be modified only if really substantial evidence is available (say, $N_{heads} \gg 50$, $N_{all} \gg 100$). By contrast, the version with $m = 1$ makes it possible to invalidate the user's opinion even with just a few experiments.

Domains with More Than Two Outcomes Although we have so far confined ourselves to a two-class domain, the same formula works just as well in multi-outcome domains. For instance, rolling a fair die can result in six different values, and we expect that the probability of obtaining, say, three points is $\pi_{three} = 1/6$. Setting our "confidence" parameter to $m = 6$, we obtain the following:

$$P_{three} = \frac{N_{three} + m\pi_{three}}{N_{all} + m} = \frac{N_{three} + 6 \cdot \frac{1}{6}}{N_{all} + 6} = \frac{N_{three} + 1}{N_{all} + 6}$$

Again, if N_{all} is so high as to render $m = 6$ and $m\pi_{three} = 1$ negligible, the formula converges to relative frequency: $P_{three} = \frac{N_{three}}{N_{all}}$; an engineer having a high confidence in his or her prior estimate, π_{three}), will choose for m a much higher value. In that event, many more experimental trials will be needed to contradict the estimates.

Limitations of m-Estimates Keep in mind that m-estimate is only as good as the values it uses. If the prior estimate is poor, the results can be disappointing. For instance, let us suppose that in the coin experiment, the prior estimate is (unrealistically) specified as $\pi_{heads} = 0.9$, with the confidence parameter set to $m = 10$. Then we obtain the following:

$$P_{heads} = \frac{N_{heads} + 9}{N_{all} + 10}$$

Using this fraction to recalculate the m-estimates from Table 2.4, we realize that, after the entire experiment (three *heads* in five trials), the probability is $P_{heads} =$

$\frac{3+9}{5+10} = \frac{12}{15} = 0.8$, surely a less plausible result than the one obtained in the case of $\pi_{heads} = 0.5$ where we got $P_{heads} = 0.57$. The reader may want to verify that the situation is less serious if $m = 1$ is used. Smaller values of m make it easier to correct poor estimates.

Mathematical Soundness We know from mathematics that the probabilities of all possible events have to sum to 1. For an experiment with N different outcomes, where P_i is the probability of the i-th outcome, this means $\sum_{i=1}^{N} P_i = 1$. It is easy to verify that Eq. 2.7 satisfies this condition for any value of m. Thus in the coin-tossing domain with two different outcomes whose prior estimates sum to 1 ($\pi_{heads} + \pi_{tails} = 1$), we derive the following (knowing that $N_{heads} + N_{tails} = N_{all}$):

$$\begin{aligned} P_{heads} + P_{tails} &= \frac{N_{heads} + m\pi_{heads}}{N_{all} + m} + \frac{N_{tails} + m\pi_{tails}}{N_{all} + m} \\ &= \frac{N_{heads} + N_{tails} + m(\pi_{heads} + \pi_{tails})}{N_{all} + m} = 1 \end{aligned}$$

The reader will easily generalize this result for any finite number of classes.

Another Benefit of m-Estimates In the problem shown in Table 2.5, we want the Bayesian classifier to classify example x. To begin with, we need to calculate the requisite conditional probabilities. Trying to do so for the positive class, however, we realize that, since the training set is so small, none of the training examples has `crust-shade=gray`, the value observed in **x**. Relative frequency gives P(`crust-shade=gray|pos`) = 0/5 = 0, and the product of individual conditional probabilities is P(x | pos) = $\prod_{i=1}^{n} P(x_i|\text{pos}) = 0$, regardless of the probabilities for the other attributes. This does not seem right.

In the case of m-estimates, this cannot happen because an m-estimate can never drop to zero. Put another way, this provides protection against the situation where a concrete attribute value does not occur for the given class (in the training set), a circumstance that would zero out the overall probability of **x** if relative frequency were used.

2.3.1 *What Have You Learned?*

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Under what circumstances is relative frequency ill-suited to estimate discrete probabilities?
- What is the impact of parameter m in Eq. 2.7? When will you prefer a high value of m, and when will you prefer smaller a one?

Table 2.5 Another reason to use m-estimates in Bayesian classification

Return to the "pies" domain from Table 1.1. Remove from the table the first example, then use the rest for the calculation of the probabilities.

```
x = [shape=circle, crust-size=thick, crust-shade=gray
      filling-size=thick, filling-shade=dark]
```

Let us first calculate the probabilities of the individual attribute values:

P(shape=circle|pos)	$= 3/5$
P(crust-size=thick|pos)	$= 4/5$
P(crust-shade=gray|pos)	$= 0/5$
P(filling-size=thick|pos)	$= 2/5$
P(filling-shade=dark|pos)	$= 3/5$

Based on these values, we obtain the following:
$P(\text{x}|\text{pos}) = \frac{3\times4\times0\times2\times3}{5^5} = 0.$

The fact that none of the five positive examples has `crust-shade=gray` thus results in the zero conditional probability $P(\text{x}|\text{pos})$. This does not happen when the probabilities are calculated by the m-estimate—which can never drop to 0.

Let us suppose that each attribute value is equally likely. For instance, there are three different shapes, and the probability of each shape is $p = 1/3$. Let us set the confidence parameter to $m = 1/p$. The conditional probabilities are then as follows:

P(shape=circle|pos)	$= \frac{3+1}{5+3} = 4/8$
P(crust-size=thick|pos)	$= \frac{4+1}{5+2} = 5/7$
P(crust-shade=gray|pos)	$= \frac{0+1}{5+3} = 1/8$
P(filling-size=thick|pos)	$= \frac{2+1}{5+2} = 3/7$
P(filling-shade=dark|pos)	$= \frac{3+1}{5+3} = 4/8$

Now we obtain the following:
$P(\text{x}|\text{pos}) = \frac{4\times5\times1\times3\times4}{8^3\times7^2} \neq 0.$

- What is the impact of the prior estimate, π_{heads}, in Eq. 2.7? How is the credibility of m-estimates affected by unrealistic values of π_{heads}? Can appropriate setting of m help alleviate the situation? Support your claim by concrete calculations.

2.4 Continuous Attributes: Probability Density Functions

So far, we have constrained our considerations to *discrete* attributes, estimating the probabilities of these attributes' individual values either by relative frequency or by m-estimate. In many applications, however, examples are described by attributes that acquire values from *continuous* domains, as is the case of `price` or `age`.

With continuous-valued attributes, relative frequency is impractical. While it is easy to establish that the chances of an engineering student being male is, say, $P_{male} = 0.7$, the probability that this student's body-weight is 184.5 pounds cannot

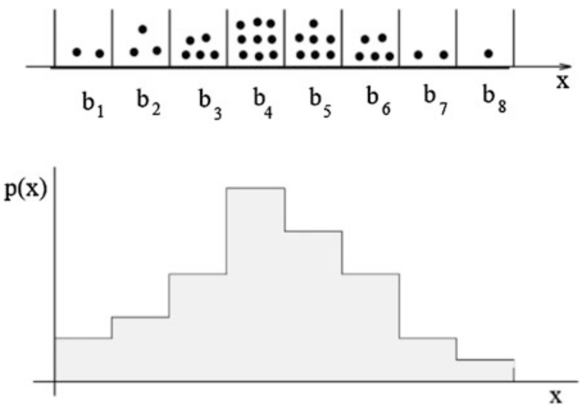


Fig. 2.2 A simple discretization method that represents each sub-interval by a separate bin. The bottom chart plots the histogram over the sub-intervals

be addressed by frequency: the number of different `weights` being infinite, the probability of any concrete value is infinitesimally small. How can this be handled?

Discretizing Continuous Attributes One possibility is to resort to the so-called *discretization*. The simplest "trick" is to split the attribute's original domain into two. For instance, we can replace the continuous-valued attribute `age` with the Boolean attribute `old` whose value is *true* for `age` $>$ 60 and *false* otherwise. Unfortunately, this means that at least part of the available information is lost: a person may be `old`, but we no longer know *how* old; nor do we know whether one `old` person is older than another `old` person.

The loss is mitigated if we divide the original domain into not two, but several intervals, say, (0, 10], ... (90, 100].[1] Suppose we provide a separate bin for each of these, and place a little black ball into the i-th bin for each training example whose value of `age` falls into the i-th interval.

In this way, we may reach a situation similar to the one depicted in Fig. 2.2. The upper part shows the bins, and the bottom part shows a step function created in the following manner: if N is the size of the training set, and N_i is the number of balls in the i-th bin, then the function's value in the i-th interval is N_i/N, the relative frequency of the i-the interval balls in the whole set. Since the area under the function is $\frac{\Sigma N_i}{N} = 1$, we have a mechanism to estimate the probability *not* of a concrete value of `age`, but rather of this value falling into the given interval.

Probability Density Function If the step function thus constructed seems too crude, we may fine-tune it by dividing the original domain into shorter—and thus more numerous—intervals, provided that the number of balls in each bin is sufficient for reliable probability estimates. If the training set is infinitely large, we can, theoretically speaking, keep reducing the lengths of the intervals until these intervals become infinitesimally short. The result of the bin-filling exercise will then no longer be a step function, but rather a continuous function, $p(x)$, such as the one

[1]We assume here that 100 is the maximum value observed in the training set.

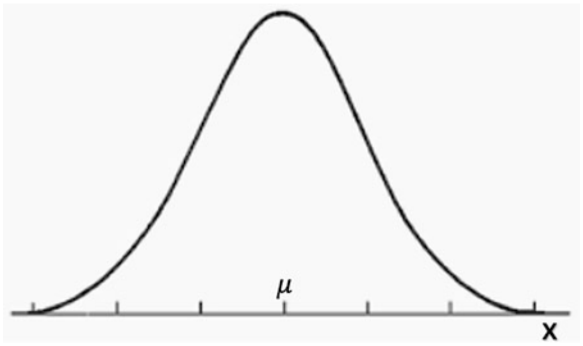


Fig. 2.3 For continuous-valued attributes, probability density function, $p(x)$, indicates the density of values in the vicinity of x. This graph shows the popular theoretical *pdf*: the Gaussian Bell Function

in Fig. 2.3. Its interpretation is obvious: a high value of $p(x)$ indicates that there are many examples with `age` close to x; conversely, a low value of $p(x)$ tells us that `age` values in the vicinity of x are rare.

Put another way, $p(x)$ is the *density* of values around x. This is why $p(x)$ is usually referred to as a *probability density function*. Engineers often prefer the acronym *pdf*.

Let us be disciplined about the notation. The probability of a discrete-valued x will be indicated by an upper-case letter, $P(x)$. By contrast, the value of a *pdf* at x will be denoted by a lower-case letter, $p(x)$. When we want to point out that the *pdf* has been created exclusively from examples belonging to class c_i, we do so by using a subscript, $p_{c_i}(x)$.

Bayes Formula for Continuous Attributes The good thing about the *pdf* is that it makes it possible to employ the Bayes formula even in the case of continuous attributes. We only replace the conditional probability $P(x|c_i)$ with $p_{c_i}(x)$, and $P(x)$ with $p(x)$. Let us begin with the trivial case where the object to be classified is described by a single continuous attribute, x. The Bayes formula then assumes the following form:

$$P(c_i \mid x) = \frac{p_{c_i}(x) \cdot P(c_i)}{p(x)} \tag{2.8}$$

Here, $P(c_i)$ is estimated by the relative frequency of class c_i in the training set, $p(x)$ is the *pdf* obtained by analyzing all training examples, and $p_{c_i}(x)$ is a *pdf* created only from training examples belonging to c_i.

Again, the denominator can be ignored because it has the same value for each class. A practical classifier will simply calculate, separately for each class, the value of the numerator, $p_{c_i}(x)P(c_i)$, and then label the object with the class for which the numerator is maximized.

Naive Bayes Revisited When facing the more realistic case where examples are described not by a single attribute but by a vector of attributes, we use the same "trick" as before: the (naive) assumption that all attributes are mutually independent.

Consider an example described by $\mathbf{x} = (x_1, \ldots, x_n)$. The *pdf* at $\mathbf{x}$ is approximated by the product of the *pdf*'s at the values of the individual attributes:

$$p_{c_j}(\mathbf{x}) = \Pi_{i=1}^{n} p_{c_j}(x_i) \tag{2.9}$$

An accomplished statistician can suggest formulas that are theoretically sounder in the sense that they do not need the naive assumption. However, higher sophistication does not guarantee success. For one thing, we may commit the sin of making precise calculations with imprecise numbers. For another, the more complicated the technique, the greater the danger of applying it incorrectly.

2.4.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- What is the probability density function, *pdf*, and what are its benefits in the context of Bayesian classification?
- Explain the discretization mechanism that helped us arrive at an informal definition of a *pdf*.
- How does the Bayes formula change in domains with continuous attributes? How is the Naive Bayes assumption used?

2.5 Gaussian "Bell" Function: A Standard *pdf*

One way to approximate a *pdf* is by the discretization technique from the previous section. Alternatively, we may choose to rely on standardized models known to work well in many realistic situations. Perhaps the most popular among these is the *Gaussian function*, named after the great German mathematician.

The Shape and the Formula Describing It The shape of the curve in Fig. 2.3 explains why it is nicknamed "bell function." The maximum is reached at the mean, $x = \mu$, and the curve slopes down gracefully with the growing distance of x from μ. It is reasonable to expect that this is a good model of the *pdf* of such variables as the body temperature where the density peaks at $x = 99.7$ degrees Fahrenheit.

Mathematically speaking, the Gaussian function is defined by the following formula where e is the base of natural logarithm:

$$p(x) = k \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.10}$$

Parameters Note that the greater the difference between x and μ, the greater the exponent's numerator, and thus the smaller the value of $p(x)$ because the exponent is negative. The numerator is squared, $(x-\mu)^2$, to make sure that the function slopes down symmetrically on both sides of the mean, μ. How steep the slope is depends on σ^2, a parameter called *variance*. Greater variance means smaller sensitivity to the difference between x and μ, and thus a "flatter" bell curve; conversely, smaller variance implies a narrower bell curve.

The task for coefficient k is to make the area under the bell function equal to 1 as required by the theory of probability. It would be relatively easy to prove that this happens when k is determined as follows:

$$k = \frac{1}{\sqrt{2\pi\sigma^2}} \tag{2.11}$$

Setting the Parameter Values To use Gaussian function when approximating $p_{c_i}(x)$ in a concrete application, we need to specify two parameters, μ and σ^2. This is easy. Suppose that class c_i is represented by m training examples. If x_i is the value of the given attribute in the i-th example, then the mean and variance, respectively, are obtained by the following formulas:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.12}$$

$$\sigma^2 = \frac{1}{m-1}\sum_{i=1}^{m}(x_i - \mu)^2 \tag{2.13}$$

In plain English, the Gaussian center, μ, is the arithmetic average of the values observed in the training examples, and the variance is the average of the squared differences between x_i and μ. Note that, when calculating variance, we divide the sum by $m - 1$, and not by m, as the reader might expect. The reason is that the formula has to compensate for the fact that μ is itself only an estimate. The variance should therefore be somewhat higher than what it would have been be if we divided by m. Of course, this matters only when the training set is small. For large m, the difference between m and $m - 1$ is negligible.

2.5.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Suggest continuous variables whose *pdf*'s can be expected to follow the Gaussian distribution.

- What parameters define the Gaussian bell function? How can we establish their values based on the training set?
- Why do we normalize the bell function? How do we do that?

2.6 Approximating PDFs with Sets of Gaussian Functions

While the bell function offers a good mechanism to approximate the *pdf* in many realistic domains, it is not a panacea. Some variables simply do not behave that way. Just consider the distribution of `body-weight` in a group that mixes grade-school children with their parents. If we create the *pdf* using the discretization method, we will observe two peaks: one for the kids, and the other for the grown-ups. There may be three peaks if it turns out that `body-weight` of fathers is distributed around a higher mean than that of the mothers. And the number of peaks can be higher still if the families come from diverse ethnic groups.

Combining Gaussian Functions In domains of this kind, a single bell function does not fit the data. But what if we combine two or more of them? If we know the diverse data subsets (e.g., children, fathers, mothers), we may simply create a separate Gaussian for each group and then superimpose the bell functions on each other. Will this solve our problem?

The honest answer is, “yes, in this specific case.” In reality, prior knowledge about diverse subgroups is rarely available. A better solution will divide the `body-weight` values into many random groups; in the extreme, we may go as far as to make each example a single-member “group” of its own and then identify a Gaussian center with this example’s `body-weight`. For m examples, this results in m bell function.

The Formula to Combine Them Suppose we want to approximate the *pdf* of a continuous attribute, x. If we denote by μ_i the value of x in the i-th example, then the *pdf* is approximated by the following sum of m Gaussian functions:

$$p(x) = k \cdot \Sigma_{i=1}^{m} e^{-\frac{(x-\mu_i)^2}{2\sigma^2}} \tag{2.14}$$

As before, the normalization constant, k, is to make sure that the area under the curve is 1. This is achieved when k is calculated as follows:

$$k = \frac{1}{m\sigma\sqrt{2\pi}} \tag{2.15}$$

If m is sufficiently high, Eq. 2.14 will approximate the *pdf* with almost arbitrary accuracy.

Numeric Example Figure 2.4 illustrates the approach using a training set consisting of $m = 3$ examples, the values of attribute x being $x_1 = 0.4, x_2 = 0.5$, and

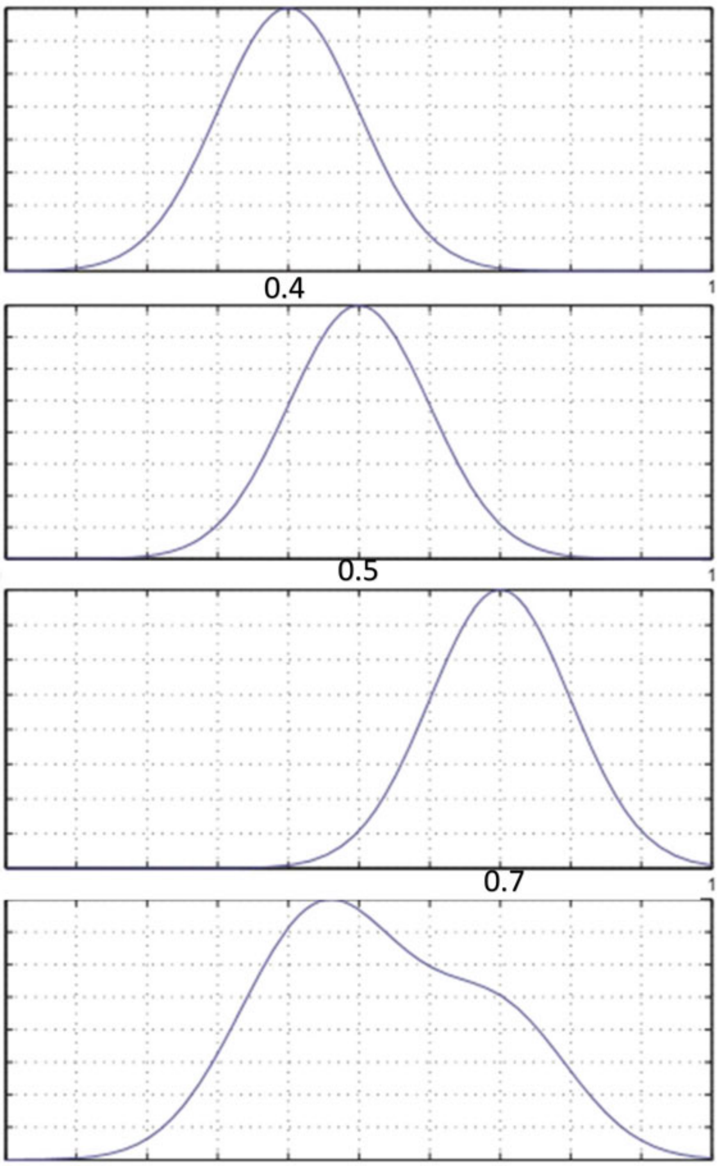


Fig. 2.4 Composing the *pdf* from three examples with the following values of attribute x: $\mu_1 = 0.4$, $\mu_2 = 0.5$, and $\mu_3 = 0.7$. The upper three charts show the contributing Gaussian functions; the bottom chart, the composition. The variance is $\sigma^2 = 1$

$x_3 = 0.7$. The upper three charts show three bell functions, each centered at one of these points, the variance always being $\sigma^2 = 1$. The bottom chart shows the composed *pdf* created by putting together Eq. 2.14 and Eq. 2.15, using the means, $\mu_1 = 0.4$, $\mu_2 = 0.5$, and $\mu_3 = 0.7$, and $\sigma^2 = 1$:

$$p(x) = \frac{1}{3\sqrt{2\pi}} \cdot [e^{-\frac{(x-0.4)^2}{2}} + e^{-\frac{(x-0.5)^2}{2}} + e^{-\frac{(x-0.7)^2}{2}}]$$

Impact of Parameter Values Practical utility of the *pdf* thus obtained (its likely success when employed by the Bayesian classifier) depends on the choice of σ^2. In Fig. 2.4, we used $\sigma^2 = 1$, but there is no guarantee that this will work in any future application. To adjust it properly, we need to understand how it affects the shape of the composite *pdf*.

Inspecting the Gaussian formula, we realize that the choice of a very small value of σ^2 causes great sensitivity to the difference between x and μ_i; the individual bell functions will be "narrow," and the resulting *pdf* will be marked by steep peaks separated by extended "valleys." Conversely, the consequence of a high σ^2 will be an almost flat *pdf*. Seeking a compromise between the two extremes, we will do well if we make σ^2 dependent on the distances between examples.

The simplest solution will use $\sigma^2 = \mu_{max} - \mu_{min}$, where μ_{max} and μ_{min} are the maximum and minimum values of μ_i, respectively. If this seems too crude, you may normalize the difference by the number of examples: $\sigma^2 = (\mu_{max} - \mu_{min})/m$. Large training sets (with high m) will then lead to smaller variations that will narrow the contributing Gaussian functions. Finally, in some domains we might argue that each of the contributing bell functions should have a variance of its own, proportional to the distance from the center of the nearest other bell function. In this case, however, we are no longer allowed to set the value of k by Eq. 2.15.

Numeric Example Table 2.6 illustrates the procedure using a very small training set and a vector to be classified. The reader is encouraged to go through all details so as to get used to the way the formulas are put together. See also the illustration in Fig. 2.5.

Domains with Too Many Examples For a training set of realistic size, it is not practical to identify each training example with one Gaussian centers—nor is it necessary. More often than not, the examples tend to be grouped in *clusters* that can be detected by *cluster analysis* techniques such as those that will be discussed in Chap. 15. Once the clusters have been found, it is reasonable to identify each Gaussian function with one cluster.

2.6.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

Table 2.6 Using Naive Bayes in domains with three continuous attributes

Consider a training set that consists of six examples, ex_1, ..., ex_6, each described by three continuous attributes, $at_1, at_2, \ldots at_3$:

Example	at_1	at_2	at_3	Class
ex_1	3.2	2.1	2.1	pos
ex_2	5.2	6.1	7.5	pos
ex_3	8.5	1.3	0.5	pos
ex_4	2.3	5.4	2.45	neg
ex_5	6.2	3.1	4.4	neg
ex_6	1.3	6.0	3.35	neg

Using the Bayes formula, we are to find the most probable class of $\mathbf{x} = (9, 2.6, 3.3)$.

Let us evaluate $p_{\text{pos}}(\mathbf{x}) \cdot P(\text{pos})$ and $p_{\text{neg}}(\mathbf{x}) \cdot P(\text{neg})$. Seeing that $P(\text{pos}) = P(\text{neg})$, we will label $\mathbf{x}$ with pos if $p_{\text{pos}}(\mathbf{x}) > p_{\text{neg}}(\mathbf{x})$ and with neg otherwise. When constructing the *pdf*'s, we follow the Naive Bayes assumption of independent attributes.

$p_{\text{pos}}(\mathbf{x}) = p_{\text{pos}}(at_1) \cdot p_{\text{pos}}(at_2) \cdot p_{\text{pos}}(at_3)$ and

$p_{\text{neg}}(\mathbf{x}) = p_{\text{neg}}(at_1) \cdot p_{\text{neg}}(at_2) \cdot p_{\text{neg}}(at_3)$

The right-hand sides are obtained by Eq. 2.14, in which we use $\sigma^2 = 1, m = 3$, and, therefore, $k = 1/\sqrt{(2\pi)^3}$. Thus for the first of the terms, we get the following:

$$p_{\text{pos}}(at_1) = \frac{1}{3\sqrt{2\pi}}[e^{-0.5(at_1-3.2)^2} + e^{-0.5(at_1-5.2)^2} + e^{-0.5(at_1-8.5)^2}]$$

Note that the values of at_1 in the positive examples are $\mu_1 = 3.2, \mu_2 = 5.2$, and $\mu_3 = 8.5$, respectively—see the exponents in the expression. The functions for the remaining five terms, obtained similarly, are plotted in the right-most column of Fig. 2.5.

Substituting into these equations the coordinates of $\mathbf{x}$, namely $at_1 = 9$, $at_2 = 3.6$, and $at_3 = 3.3$, will give us the following:

$p_{\text{pos}}(\mathbf{x}) = 0.0561 \times 0.0835 \times 0.0322 = 0.00015$

$p_{\text{neg}}(\mathbf{x}) = 0.0023 \times 0.0575 \times 0.1423 = 0.00001$

Observing that $p_{pos}(\mathbf{x}) > p_{neg}(\mathbf{x})$ (and knowing that both classes are equally represented, $P(\text{pos}) = P(\text{neg}) = 0.5$), we label $\mathbf{x}$ with class pos.

- Under what circumstances will the Gaussian bell function poorly approximate the *pdf*?
- Why does the composite *pdf* have to be normalized by k?
- How do we establish the centers and variances of the individual bell functions?

2.7 Summary and Historical Remarks

- Bayesian classifiers calculate the product $P(\mathbf{x}|c_i)P(c_i)$ separately for each class, c_i, and then label $\mathbf{x}$ with the class where this product has the highest value.
- The main problem is how to calculate the probability, $P(\mathbf{x}|c_i)$. Most of the time, the job is simplified by the assumption that the attributes are mutually independent, in which case $P(\mathbf{x}|c_i) = \prod_{j=1}^{n} P(x_j|c_i)$, where n is the number of attributes.

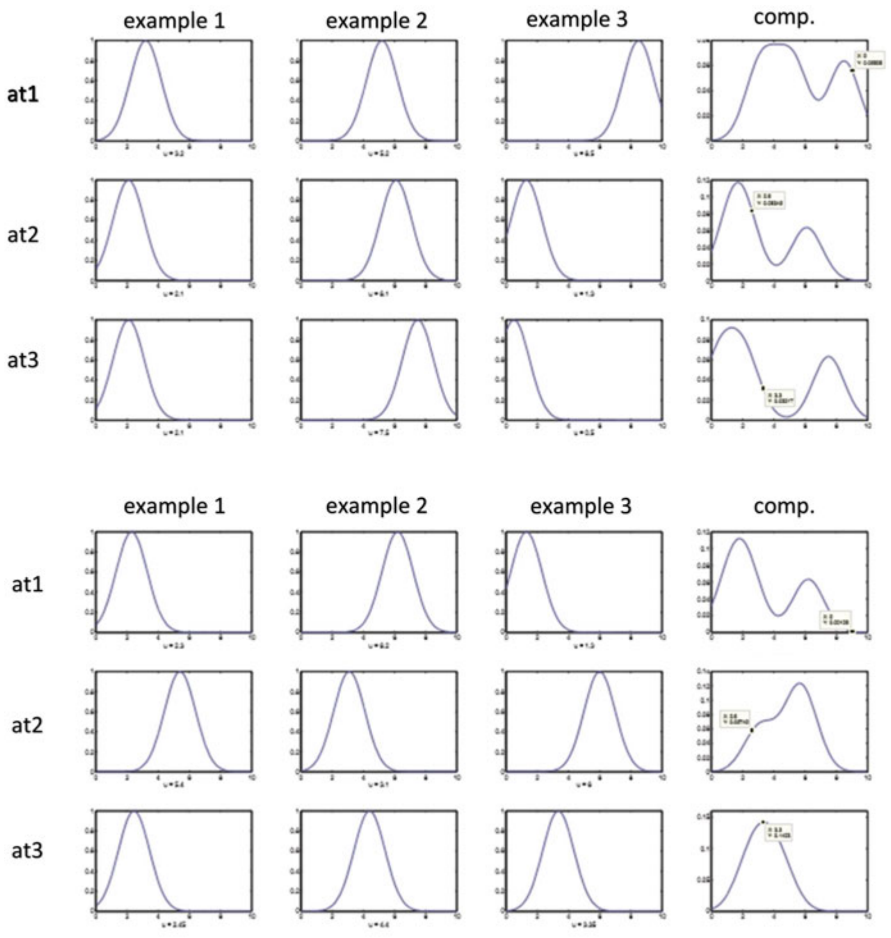


Fig. 2.5 Composing the *pdf*'s separately for the positive and negative class (with $\sigma^2 = 1$). Each row represents one attribute, and each of the left three columns represents one example. The rightmost column shows the composed *pdf*'s

- The so-called m-estimate makes it possible to take advantage of a user's prior idea about an event's probability. This comes handy in domains with small training sets where relative frequency is unreliable.
- In domains with continuous attributes, the role of discrete probability, $P(\mathbf{x}|c_i)$, is taken over by $p_{c_i}(\mathbf{x})$, the probability density function, *pdf*. Otherwise, the procedure is the same: the example is labeled with the class that maximizes the product, $p_{c_i}(\mathbf{x})P(c_i)$.
- The concrete shape of the *pdf* is approximated by discretization, or by the use of standardized *pdf*s, or by the sum of Gaussian functions.

- The estimates of probabilities are far from perfect, but the results are often satisfactory even when rigorous theoretical assumptions are not satisfied.

Historical Remarks The first papers to propose the use of Bayesian decision theory for classification purposes were Neyman and Pearson (1928) and Fisher (1936), but the paradigm gained momentum only with the advent of the computer, when it was advocated by Chow (1957). The first to use the assumption of independent attributes was Good (1965). The idea of approximating *pdf*'s by the sum of bell functions comes from Parzen (1962).

When provided with perfect information about the probabilities, the Bayesian classifier is guaranteed to provide the best possible classification accuracy. This is why it is sometimes used as a reference to which the performance of other approaches is compared.

2.8 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

2.8.1 Exercises

1. A coin tossed three times came up *heads, tails*, and *tails*, respectively. Calculate the m-estimate for these outcomes, using $m = 3$ and $\pi_{heads} = \pi_{tails} = 0.5$.
2. Consider the following training examples, described by three attributes, x_1, x_2, x_3, and labeled by classes c_1 and c_2.

x_1	x_2	x_3	Class
2.1	0.2	3.0	c_1
3.3	1.0	2.9	c_1
2.7	1.2	3.4	c_1
0.5	5.3	0.0	c_2
1.5	4.7	0.5	c_2

Using these data, do the following:

(a) Assuming that the attributes are mutually independent, approximate the following probability density functions: $p_{c_1}(\mathbf{x})$, $p_{c_2}(\mathbf{x})$, $p(\mathbf{x})$. Hint: use the idea of superimposed bell functions.

(b) Using the *pdf*'s from the previous step, decide whether $\mathbf{x} = [1.4, 3.3, 3.0]$ should belong to c_1 or c_2.

2.8.2 Give It Some Thought

1. How would you employ m-estimate in a domain with three possible outcomes, $[A, B, C]$, each with the same prior probability estimate, $\pi_A = \pi_B = \pi_C = 1/3$? What if you trust your expectations of A while not being so sure about B and C? Is there a way to reflect this circumstance in the value of the parameter m?
2. Explain under which circumstances the accuracy of probability estimates benefits from the assumption that attributes are mutually independent. Explain the advantages and disadvantages.
3. How would you calculate the probabilities of the output classes in a domain where some attributes are Boolean, others discrete, and yet others continuous? Discuss the possibilities of combining different approaches.

2.8.3 Computer Assignments

1. Machine-learning researchers often test their algorithms on publicly available benchmark domains. A large repository of such domains can be found at the following address: `www.ics.uci.edu/~mlearn/MLRepository.html`. Take a look at these data and see how they differ in the numbers of attributes, types of attributes, sizes, and so on.
2. Write a computer program that uses the Bayes formula to calculate the class probabilities in a domain where all attributes are discrete. Apply this program to our "pies" domain.
3. For the case of continuous attributes, write a computer program that accepts the training examples in the form of a table such as the one from Exercise 3 above. Based on these, the program approximates the *pdf*s and then uses them to determine the class labels of future examples.
4. Apply this program to a few benchmark domains from the UCI repository (choose from among those where all attributes are continuous) and observe that the program succeeds in some domains better than in others.

Chapter 3
Similarities: Nearest-Neighbor Classifiers

Two trees that look very much alike probably represent the same species; likewise, it is common that patients complaining of similar symptoms suffer from the same disease. Similar objects often belong to the same class—an observation underlying another popular approach to classification: when asked to determine the class of object **x**, find the training example most similar to it, and then label **x** with this similar example's class.

The chapter explains how to evaluate example-to-example similarities, presents concrete mechanisms that use these similarities for classification purposes, compares the performance of this approach with that of Bayesian classifiers, discusses sensitive aspects of this paradigm, such as scaling, data sparseness, and sensitivity to harmful attributes and harmful examples, and introduces methods to overcome its shortcomings.

Nearest-neighbor classifiers are somewhat outdated, and as such are rarely used. The reason why this book dedicates to them an entire chapter is educational: the simplicity of this paradigm makes it easy to explore various complications and obstacles that are typical also of many other machine-learning approaches.

3.1 The *k*-Nearest-Neighbor Rule

How do we establish that a certain object is more similar to **x** than to **y**? Some may doubt that this is at all possible. Is giraffe more similar to horse than to zebra? Questions of this kind raise suspicion. Too many arbitrary and subjective factors have to be considered when looking for an answer.

Similarity of Attribute Vectors The machine-learning task formulated in the previous chapters keeps the situation relatively simple. Rather than real objects, the classifier compares their attribute-based descriptions. Thus in the toy domain from Chap. 1, the similarity of two pies can be established by counting the attributes in

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_3

Table 3.1 Counting the numbers of differences between pairs of discrete-attribute vectors. Of the twelve training examples, $\mathtt{ex}_5$ is the one most similar to **x**

		Crust		Filling			
Example	Shape	Size	Shade	Size	Shade	Class	# differences
x	Square	Thick	Gray	Thin	White	?	–
$\mathtt{ex}_1$	Circle	Thick	Gray	Thick	Dark	pos	3
$\mathtt{ex}_2$	Circle	Thick	White	Thick	Dark	pos	4
$\mathtt{ex}_3$	Triangle	Thick	Dark	Thick	Gray	pos	4
$\mathtt{ex}_4$	Circle	Thin	White	Thin	Dark	pos	4
$\mathtt{ex}_5$	Square	Thick	Dark	Thin	White	pos	1
$\mathtt{ex}_6$	Circle	Thick	White	Thin	Dark	pos	3
$\mathtt{ex}_7$	Circle	Thick	Gray	Thick	White	neg	2
$\mathtt{ex}_8$	Square	Thick	White	Thick	Gray	neg	3
$\mathtt{ex}_9$	Triangle	Thin	Gray	Thin	Dark	neg	3
$\mathtt{ex}_{10}$	Circle	Thick	Dark	Thick	White	neg	3
$\mathtt{ex}_{11}$	Square	Thick	White	Thick	Dark	neg	3
$\mathtt{ex}_{12}$	Triangle	Thick	White	Thick	Gray	neg	4

which they differ: the fewer the differences, the greater the similarity. The first row in Table 3.1 gives the attribute values of object **x**. For each of the twelve training examples that follow, the right-most column specifies the number of differences in the attribute values of the given example and **x**. The smallest value being found in the case of $\mathtt{ex}_5$, we conclude that this is the training example most similar to **x**, and **x** should thus be labeled with pos, the class of $\mathtt{ex}_5$.

In Table 3.1, all attributes are discrete, but dealing with continuous attributes is just as easy. Since each example can be represented by a point in an n-dimensional space, we can use the Euclidean distance or some other geometric formula (Section 3.2 will have more to say on this topic); and again, the smaller the distance, the greater the similarity. This, by the way, is how the *nearest-neighbor classifier* got its name: the training example with the smallest distance from **x** in the instance space is, geometrically speaking, **x**'s nearest neighbor.

From a Single Neighbor to k Neighbors In noisy domains, the single nearest neighbor cannot be trusted. What if its class label is incorrect due to noise? A more robust approach will identify not one but several nearest neighbors, and let them vote. This is the essence of the k-NN classifier, where k is the number of the voting neighbors—usually a user-specified parameter. The pseudo-code in Table 3.2 summarizes the approach.

Note that, say, a 4-NN classifier, when applied to a 2-class domain, may result in a situation where two neighbors are positive and two negative. In this event, it is not clear how to classify. Ties of this kind are avoided by using for k an odd number.

In domains with more than two classes, however, an odd number of nearest neighbors does not help. For instance, a 7-NN classifier can realize that three neighbors belong to class C_1, three neighbors to C_2, and one neighbor to C_3. The

Table 3.2 The simplest version of the *k*-NN classifier

Suppose we have a mechanism to evaluate the similarity of attribute vectors. Let **x** denote the object whose class we want to determine.

1. In the training set, identify the k nearest neighbors of **x** (the k examples most similar to **x**).
2. Let c_i be the class most frequently found among these k nearest neighbors.
3. Label **x** with c_i.

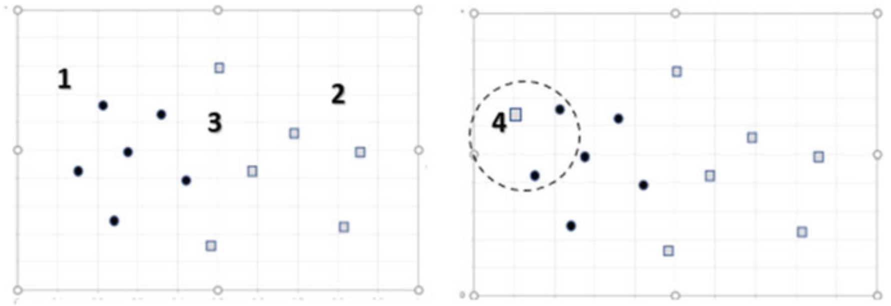


Fig. 3.1 Objects **1** and **2** find themselves deep in the "circles" and "squares" areas, respectively, and are thus easy to classify. Object **3**, finding itself in the borderline region between the two classes, and its class is thus uncertain. In the class-noisy domain on the right, the 1-NN classifier will misclassify object **4**, but the mistake is corrected if the 3-NN classifier is used

engineer implementing this classifier must not forget to specify a mechanism that chooses between the tied classes.

Illustration Certain aspects of this paradigm's behavior can be illuminated by the use of a fictitious domain where examples are described by two numeric attributes, a situation easy to visualize. The two graphs in Fig. 3.1 show several positive and negative training examples and also some objects (the big black digits) whose classes the *k*-NN classifier is to determine. The reader can see that objects **1** and **2** are surrounded by examples that all belong to the same class, which makes their classification straightforward. On the other hand, object **3** is located in the "no man's land" between the positive and negative regions so that even a small amount of attribute noise can send it to either side. The classification of such *borderline examples* is of course unreliable.

In the right-hand part of the picture, object **4** finds itself deep in the "black circles" region, but class noise has mislabeled its nearest neighbor in the training set as a `square`. Whereas the 1-NN classifier will go wrong, here, the 3-NN classifier will classify correctly because the other two neighbors will outvote the single offender.

3.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How can we measure example-to-example similarity in domains where all attributes are discrete and how in domains where they are continuous?
- Under what circumstances will the k-NN classifier (with $k > 1$) outperform the 1-NN classifier and why?
- Explain why, in 2-class domains, the k in the k-NN classifier should be an odd number. Why is this unimportant in multi-class domains?
- How does attribute noise affect the classification of borderline examples? What is the impact of class-label noise?

3.2 Measuring Similarity

As mentioned earlier, a natural way to identify the nearest neighbor of some $\mathbf{x}$ is to use the geometrical distances of $\mathbf{x}$ from the training examples. Figure 3.1 shows a two-dimensional domain where the distances can easily be measured by a ruler—but the ruler surely cannot be used if there are more than three attributes. In that event, we need a mathematical formula.

Euclidean Distance In a two-dimensional space, a plane, the geometric distance between two points, $\mathbf{x} = (x_1, x_2)$ and $\mathbf{y} = (y_1, y_2)$, is measured by the Pythagorean theorem as illustrated in Fig. 3.2: $d(\mathbf{A}, \mathbf{B}) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$. The following formula generalizes this to n-dimensional domains: the *Euclidean distance* between $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$:

$$d_E(\mathbf{x}, \mathbf{y}) = \sqrt{\Sigma_{i=1}^{n} (x_i - y_i)^2} \tag{3.1}$$

The use of this metric in k-NN classifiers is illustrated in Table 3.3 where the training set consists of four examples described by three numeric attributes.

More General Formulation The reader has noticed that the term under the square root symbol is the sum of the squared distances along the individual attributes.[1] Mathematically, this is expressed as follows:

$$d_M(\mathbf{x}, \mathbf{y}) = \sqrt{\Sigma_{i=1}^{n} d(x_i, y_i)} \tag{3.2}$$

[1]One benefit of the differences being squared, and thus guaranteed to be positive, is that negative differences, $x_i - y_i < 0$, will not be subtracted from positive differences, $x_i - y_i > 0$.

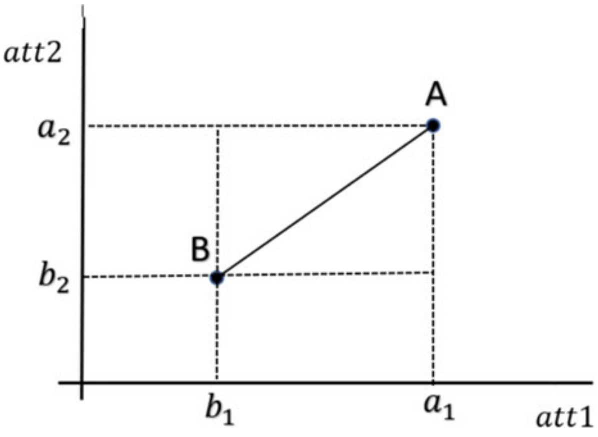


Fig. 3.2 Euclidean distance between two points in a two-dimensional space is the length of the corresponding triangle's hypotenuse

Table 3.3 The nearest-neighbor principle in a three-dimensional Euclidean space

Using the following training set of four examples described by three numeric attributes, determine the class of $\mathbf{x} = [2, 4, 2]$.

		Distance between $\texttt{ex}_i$ and [2,4,2]
$\texttt{ex}_1$	{[1,3,1],pos}	$\sqrt{(2-1)^2 + (4-3)^2 + (2-1)^2} = \sqrt{3}$
$\texttt{ex}_2$	{[3,5,2],pos}	$\sqrt{(2-3)^2 + (4-5)^2 + (2-2)^2} = \sqrt{2}$
$\texttt{ex}_3$	{[3,2,2],neg}	$\sqrt{(2-3)^2 + (4-2)^2 + (2-2)^2} = \sqrt{5}$
$\texttt{ex}_4$	{[5,2,3],neg}	$\sqrt{(2-5)^2 + (4-2)^2 + (2-3)^2} = \sqrt{14}$

Calculating Euclidean distances between **x** and all training examples, we realize that **x**'s nearest neighbor is $\texttt{ex}_2$. Its label being pos, the 1-NN classifier returns the positive label.

The same result is obtained by the 3-NN classifier because two of **x**'s three nearest neighbors ($\texttt{ex}_1$ and $\texttt{ex}_2$) are positive, and only one ($\texttt{ex}_3$) is negative

This formula makes it possible to calculate the differences in domains with a mixture of discrete and continuous attributes (in the symbol, $d_M(\mathbf{x}, \mathbf{y})$, the <u>m</u>ixture is indicated by the subscript, M). In the term under the square root, we typically use $d(x_i, y_i) = (x_i - y_i)^2$ for continuous attributes. For discrete attributes, we often put $d(x_i, y_i) = 0$ if $x_i = y_i$ and $d(x_i, y_i) = 1$ if $x_i \neq y_i$.

Note that if all attributes are continuous, the formula reduces to Euclidean distance, and if the attributes are all discrete, the formula simply specifies the number of attributes in which the two vectors differ. In purely Boolean domains, where for any attribute only the values *true* or *false* are permitted (let us abbreviate these values as t and f, respectively), this latter case is called the *Hamming distance*, d_H. For instance, the Hamming distance between the vectors $\mathbf{x} = (t, t, f, f)$ and $\mathbf{y} = (t, f, t, f)$ is $d_H(\mathbf{x}, \mathbf{y}) = 2$. In general, however, Eq. 3.2 is meant for domains where examples are described by a mixture of discrete and continuous attributes.

Attribute-to-Attribute Distances Can Be Misleading We must be careful not to use Formula 3.2 mechanically, ignoring the specific aspects of a given domain. Let us briefly discuss two circumstances that make it is easy for us to go wrong.

Suppose our examples are described by three attributes, `size`, `price`, and `season`. Of these, the first two are obviously continuous and the last discrete. If $\mathbf{x} = (2, 1.5, \texttt{summer})$ and $\mathbf{y} = (1, 0.5, \texttt{winter})$, then Eq. 3.2 gives the following distance:

$$d_M(\mathbf{x}, \mathbf{y}) = \sqrt{(2-1)^2 + (1.5-0.5)^2 + 1} = \sqrt{3}$$

Let us first consider the third attribute: since `summer` is different from `winter`, our earlier considerations indicate that $d(\texttt{summer}, \texttt{winter}) = 1$. In reality, however, the difference between `summer` and `winter` is sometimes deemed greater than the difference between, say, `summer` and `fall`, which are "neighboring seasons." Another line of reasoning may stipulate that `spring` and `fall` are more similar to each other than `summer` and `winter`—at least as far as weather is concerned. This shows that the two distance values, 0 and 1, will not suffice, here. Intermediate values should perhaps be considered, the concrete choice depending on the specific needs of the given application. The engineer who does not pay attention to factors of this kind may face disappointment.

Mixing continuous and discrete attributes can be risky in another way. A thoughtless application of Eq. 3.2 can result in a situation where the difference between two `sizes` (e.g., $\texttt{size}_1 = 1$ and $\texttt{size}_1 = 12$, which means that $d(\texttt{size}_1, \texttt{size}_2) = 11^2 = 121$) can totally dominate the difference between two `seasons` (which, in the baseline version, could not exceed 1). This observation is closely related to the problem of *scaling* discussed in the next section.

General View of Distance Metrics The reader is beginning to see that the issue of similarity is far from trivial. Apart from Eq. 3.2, quite a few other formulas have been suggested, some fairly sophisticated.[2] While it is good to know they exist, we will not examine them here for fear of digressing too far from the main topic. Suffice it to say that any distance metric has to satisfy the following requirements:

1. Distance can never be negative.
2. Distance between two identical vectors, $\mathbf{x} = \mathbf{y}$, is zero.
3. Distance from $\mathbf{x}$ to $\mathbf{y}$ is the same as distance from $\mathbf{y}$ to $\mathbf{x}$.
4. Distance must satisfy triangular inequality: $d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z}) \geq d(\mathbf{x}, \mathbf{z})$.

3.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

[2]Among these, perhaps the best known are polar distance, Minkowski's metric, and the Mahalanobis distance.

- What is the Euclidean distance, and what is the Hamming distance? In what domains can they be used? How is distance related to similarity?
- Write down the distance formula for domains where examples are described by a mixture of continuous and discrete attributes. Discuss the difficulties that complicate its straightforward application in practice.
- What fundamental properties have to be satisfied by any method of measuring distances?

3.3 Irrelevant Attributes and Scaling Problems

The reader now understands the principles of the k-NN classifier well enough to be able to write a computer program that implements it. Caution is called for, though. When applied mechanically, the tool may disappoint, and we have to understand why this may happen.

The philosophy underlying this paradigm is telling us that "objects are similar if the geometric distance between the vectors describing them is small." This said we know that the geometric distance is sometimes misleading. The following two cases are typical.

Irrelevant Attributes It is not true that all attributes are created equal. From the perspective of machine learning, some are *irrelevant* in the sense that their values have nothing to do with the example's class—and yet they affect the geometric distance between vectors.

A simple illustration will clarify the point. In the training set from Fig. 3.3, the examples are characterized by two numeric attributes: `body-temperature` (horizontal axis) and `shoe-size` (vertical axis). Suppose the k-NN classifier is to classify object **1** as healthy (`pos`) or sick (`neg`).

All positive examples find themselves in the shaded area delimited by two critical points along the "horizontal" attribute: temperatures exceeding the maximum

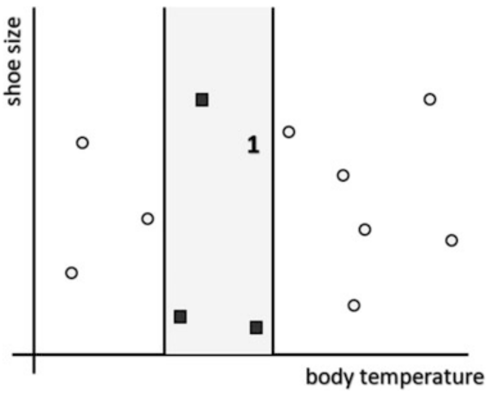


Fig. 3.3 Shoe size is not indicative of health, but it does affect geometric distances. Object **1** is in the region of healthy patients (squares), but its nearest neighbor is in the region of sick patients (circles)

indicate fever, and those below the minimum indicate hypothermia. As for the "vertical" attribute, though, we see that the positive and negative examples alike are distributed along its entire domain, `show-size` not being able to affect a person's health. The object we want to classify is in the highlighted region, and by common sense it should be labeled as positive—despite the fact that its nearest neighbor happens to be negative.

Lesson If we use only the first attribute, the Euclidean distance between the two examples is $d_E(x, y) = \sqrt{(x_1 - y_1)^2} = |x_1 - y_1|$. If both attributes are used, the Euclidean distance is $d_E(\mathbf{x}, \mathbf{y}) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$. If the second attribute is irrelevant, then the term $(x_2 - y_2)^2$ is superfluous—and yet it can modify k-NN's notion of similarity. This is what occurred in Fig. 3.3, and this is why object **1** was misclassified.

How much damage irrelevant attributes may cause depends on how many of them are used. In a domain with hundreds of attributes, of which only a single one is irrelevant, we are fairly safe: one lonely culprit is unlikely to distort the distances in any meaningful way. Things get worse, however, if the percentage of irrelevant attributes increases. In the extreme, if the vast majority of the attributes have nothing to do with the class we want to recognize, the geometric distance becomes virtually meaningless, and the classifier loses any practical value.

Scales of Attributes Suppose we want to evaluate the similarity of two examples, $\mathbf{x} = (t, 0.2, 254)$ and $\mathbf{y} = (f, 0.1, 194)$, described by three attributes, of which the first is Boolean, the second is continuous with values from [0, 1], and the third is continuous with values from [0, 1000]. Equation 3.2 calculates the distance between $\mathbf{x}$ and $\mathbf{y}$ as follows:

$$d_M(\mathbf{x}, \mathbf{y}) = \sqrt{(1-0)^2 + (0.2-0.1)^2 + (254-194)^2}$$

Inspecting this expression, we notice that the third term completely dominates, reducing the other two to insignificance. No matter how we modify their values within their ranges, the overall distance, $d_M(\mathbf{x}, \mathbf{y})$, will hardly be affected.

The situation is easy to rectify. If we divide, in the training set, all values of the third attribute by 1000, thus "squeezing" its range to [0, 1], the impacts of all attributes will become more balanced. Of course, we must not forget to divide the third attribute's value by the same 1000 in any future object we decide to classify.

We have convinced ourselves that the scales of the attribute values can strongly affect k-NN's behavior.

Another Aspect of Attribute Scales Consider the following two training examples, $\texttt{ex}_1$ and $\texttt{ex}_2$, and the object $\mathbf{x}$ whose class we want to determine:

$\texttt{ex}_1 = [(10, 10), \texttt{pos})]$
$\texttt{ex}_2 = [(20, 0), \texttt{neg})]$
$\mathbf{x} = (32, 20)$

The distances are $d_M(\mathbf{x}, \texttt{ex}_1) = \sqrt{584}$ and $d_M(\mathbf{x}, \texttt{ex}_2) = \sqrt{544}$. The latter being smaller, 1-NN will label **x** as negative. Suppose, however, that the physical meaning of the second attribute is temperature measured in Centigrade. If we decide to use Fahrenheit, the three vectors will change into the following:

$\texttt{ex}_1 = [(10, 50), \texttt{pos})]$
$\texttt{ex}_2 = [(20, 32), \texttt{neg})]$
$\mathbf{x} = (32, 68)$

Now the distances are $d_M(\mathbf{x}, \texttt{ex}_1) = \sqrt{808}$ and $d_M(\mathbf{x}, \texttt{ex}_2) = \sqrt{1440}$—and we see that now it is the first distance that is smaller; 1-NN will therefore classify **x** as positive. This seems a bit silly because the examples are still the same except that we chose different units for `temperature`. The classifier's verdict has changed because it is based on numeric calculations, not physical interpretations.

Normalizing the Attributes One way out of some scale-related troubles is to *normalize* the attributes: to rescale them so that all their values fall into the same unit interval, [0, 1]. Perhaps the simplest way of doing so is to identify, for the given attribute, its maximum (MAX) and minimum (MIN) and then replace each value, x, of this attribute using the following formula:

$$x = \frac{x - MIN}{MAX - MIN} \tag{3.3}$$

Numeric Example A simple illustration will show us how this works. Suppose that, in the training set consisting of five examples, the given attribute acquires the following values:

$$[7, 4, 25, -5, 10]$$

We see that $MIN = -5$ and $MAX = 25$. Subtracting MIN from each of the five values will result in the following:

$$[12, 9, 30, 0, 15]$$

The reader can see that the "new minimum" is 0 and the "new maximum" is $MAX - MIN = 25 - (-5) = 30$. Dividing the obtained values by $MAX - MIN$, we reach a situation where all the values fall into [0, 1]:

$$[0.4, 0.3, 1, 0, 0.5]$$

One Potential Weakness of Normalization Normalization reduces error rate in many practical applications, especially if the sizes of the domains of the original attributes vary significantly. The downside is that this may distort the examples' descriptions. Moreover, the pragmatic decision to make all values fall between 0 and

1 may not even be adequate. For instance, if the difference between `summer` and `fall` is 1, it will always be bigger than, say, the difference between two normalized body temperatures. Whether this matters or not is up to the engineer's common sense assisted by his or her experience (assisted perhaps by experimentation).

Normalize Also Attributes in Future Examples A typical mistake made by beginners in their first attempts is not to apply the same normalization formula to testing examples. Inevitably, if the same attribute has domain [0.100] in the testing example but [0, 1] in the training set, similarity is destroyed.

3.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Why do irrelevant attributes impair the k-NN classifier's performance? How is the performance affected by different percentages of irrelevant attributes?
- Explain the basic problems pertaining to attribute scaling. Describe a simple approach to normalization.
- Under what circumstances will normalization be misleading?

3.4 Performance Considerations

The k-NN technique is easy to implement in a computer program, and its behavior is easy to understand. But is there a reason to believe that its classification performance is good enough?

1-NN Versus Ideal Bayes The ultimate yardstick by which to assess any classifier's success is the Bayesian formula. If the probabilities and *pdf*'s employed in the Bayesian classifier are known with absolute accuracy, then this classifier—let us call it *Ideal Bayes*—exhibits the lowest error rate theoretically achievable on the given (noisy) data. It would be reassuring to realize that the k-NN paradigm does not trail too far behind.

The question was subjected to rigorous mathematical analysis, and here are the results. Figure 3.4 shows the comparison under such idealized circumstances as infinitely large training sets filling the instance space with infinite density. The solid curve represents the two-class case where each example is either positive or negative. We can see that if the error rate of *Ideal Bayes* is 5%, the error rate of the 1-NN classifier (vertical axis) is 10%. With the growing amount of noise, the difference between the two classifiers decreases, only to disappear when *Ideal Bayes* reaches 50% error rate—in which event, of course, the labels of the training examples are virtually random, and any attempt at automated classification is futile.

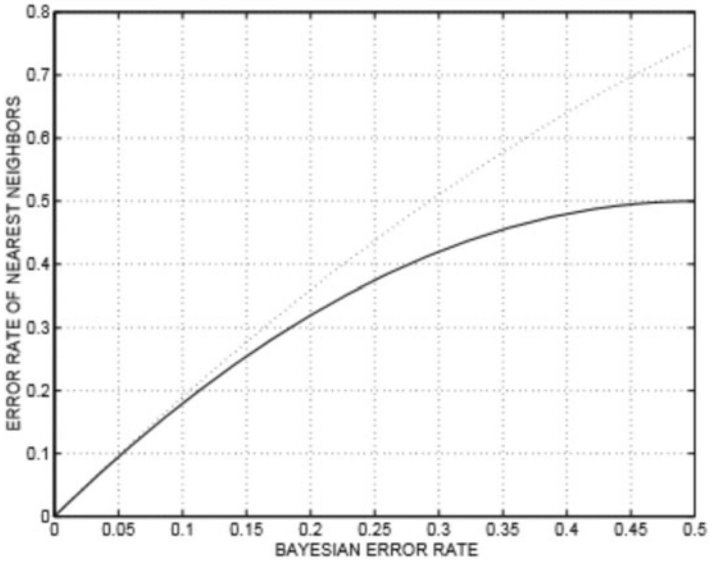


Fig. 3.4 Theoretical error rate of 1-NN compared to that of *Ideal Bayes*

The situation is only slightly better in multi-class domains, represented in the graph by the dotted curve. Again, *Ideal Bayes* outperforms the 1-NN classifier by a comfortable margin.

Increasing the Number of Neighbors From the perspective of the 1-NN classifier, the testimony of Fig. 3.4 is discouraging. But then, we know that the classifier's performance might improve when we use the more general k-NN (for $k > 1$), where some of the noisy nearest neighbors are outvoted by better-behaved ones. Does mathematics support to this intuition?

The answer is yes. Under the abovementioned ideal circumstances, the error rate has been proved to decrease with the growing k and converges to that of *Ideal Bayes* for $k \to \infty$. At least in theory, then, the performance of the nearest-neighbor classifier is able to reach the maximum.

Practical Limitations The engineer's world hardly ever lives up to theoretical expectations. In a realistic application, the training examples will but sparsely populate the instance space, and increasing the number of voting neighbors can be counterproductive. More often than not, the error rate *will* improve with the growing k, but only up to a certain point beyond which it starts worsening again in a manner indicated in Fig. 3.5, where the horizontal axis represents the values of k, and the vertical axis represents the error rate measured on an independent testing set.

Here is the interpretation of the curve's shape. Some of the more distant nearest neighbors may already be too different to be deemed similar. As such, they only mislead the classifier. Consider the extreme: if the training set contains 25 training

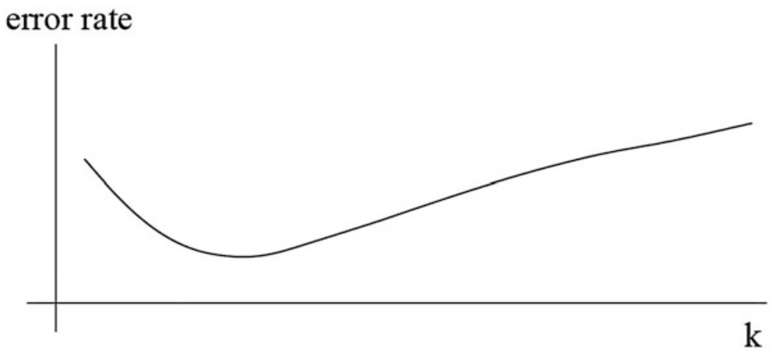


Fig. 3.5 With the growing number of voting neighbors, the error rate of k-NN decreases until it reaches a minimum from which it starts growing again

examples, then the 25-NN classifier simply labels any object with the class most common in the training data.[3]

Curse of Dimensionality We now understand that some of the nearest neighbors may not be sufficiently similar to **x** to deserve a vote. This often happens in domains with a great many attributes. Suppose that the values of each attribute are confined to the unit-length interval, [0, 1]. Using the Pythagorean theorem, we could easily show that the maximum Euclidean distance in the n-dimensional space defined by these attributes is $d_{MAX} = \sqrt{n}$. For $n = 10^4$ (a reasonable number in, say, text categorization), this means $d_{MAX} = 100$. The reader may find this surprising, given that no attribute value can exceed 1. Yet it is a mathematical fact that explains why the examples tend to be sparse unless the training set is really very large.

This is sometimes referred to as the *curse of dimensionality*: as we increase the number of attributes, the number of training examples needed to fill the instance space with adequate density grows very fast, perhaps so fast as to render the nearest-neighbor paradigm impractical.

Conclusion Although the ideal k-NN classifier is capable of reaching the performance of *Ideal Bayes*, the engineer has to be aware of the practical limitations of both approaches. *Ideal Bayes* is unrealistic in the absence of perfect knowledge of the probabilities and *pdf*'s. On the other side, k-NN is prone to suffer sparse data, irrelevant attributes, and inappropriate attribute scaling. The concrete choice depends on the specific requirements of the given application.

[3]The optimal value of k (the one with minimum error rate) can be found experimentally.

3.4.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How does the performance of k-NN compare to that of *Ideal Bayes*? Summarize this separately for $k = 1$ and $k > 1$. What theoretical assumptions do the two paradigms rely on?
- How will the performance of the k-NN classifier depend on the growing values of k? What is the difference between theory and practice?
- What is understood by the *curse of dimensionality*?

3.5 Weighted Nearest Neighbors

So far, the voting mechanism has been democratic in the sense that each nearest neighbor has the same vote. But while this seems appropriate, classification performance often improves if democracy is reduced.

Here is why. In Fig. 3.6, the task is to determine the class of object **1**. Since three of the nearest neighbors are squares and only two circles, the 5-NN classifier decides the object is `square`. However, a closer look reveals that the three square neighbors are quite distant from **1**, so much so that they perhaps should not have the same impact as the two circles in the object's immediate vicinity. After all, we want to adhere to the requirement that k-NN should classify based on similarity—and more distant neighbors are less similar than closer ones.

Weighted Nearest Neighbors Domains of this kind motivate the introduction of *weighted voting* in which the weight of each neighbor depends on its distance from the object: the closer the neighbor, the greater its impact.

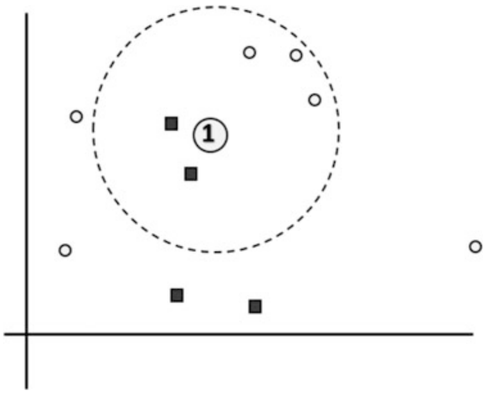


Fig. 3.6 In 5-NN, the testimony of the two *squares*, which are very close to the classified object, should outweigh the testimony of the three more distant *circles*

Let us denote the weights as $w_1, \ldots, w_k$. The *weighted k-NN* classifier sums up the weights of those neighbors that recommend the positive class (let the result be denoted by Σ^+) and then sums up the weights of those neighbors that support the negative class (Σ^-). The final verdict depends on which is higher: if $\Sigma^+ > \Sigma^-$, then the object is deemed positive; otherwise, it is labeled as negative. Generalization to domains with $n > 2$ classes is straightforward.

For illustration, suppose the positive label is found in neighbors with weights 0.6 and 0.7, respectively, and the negative label is found in neighbors with weights 0.1, 0.2, and 0.3. Weighted k-NN will choose the positive class because the combined weight of the positive neighbors, $\Sigma^+ = 0.6 + 0.7 = 1.3$, is greater than that of the negative neighbors, $\Sigma^- = 0.1 + 0.2 + 0.3 = 0.6$. Just as in Fig. 3.6, the more frequent negative neighbors are outvoted by the less frequent positive neighbors because the latter are closer (and thus more similar) to the object we want to classify.

Concrete Formula Let us introduce a simple mechanism to calculate the weights. Suppose the k neighbors are ordered by their distances, $d_1, \ldots, d_k$, from object $\mathbf{x}$ so that d_1 is the smallest distance and d_k is the greatest distance. The weight of the i-th closest neighbor is calculated as follows:

$$w_i = \begin{cases} \frac{d_k - d_i}{d_k - d_1}, & d_k \neq d_1 \\ 1 & d_k = d_1 \end{cases} \tag{3.4}$$

The weights thus obtained will range from 0 for the most distant neighbor to 1 for the closest one. The approach thus actually considers only $k - 1$ neighbors (because $w_k = 0$). Of course, this makes sense only for $k > 3$. If we used $k = 3$, then only two neighbors would really participate, and the weighted 3-NN classifier would degenerate into the 1-NN classifier.

Table 3.4 illustrates the procedure using a simple toy domain.

Table 3.4 Illustration of the weighted nearest-neighbor rule

Let the weighted 5-NN classifier determine the class of object $\mathbf{x}$. Let the distances between $\mathbf{x}$ and its five nearest neighbors be $d_1 = 1, d_2 = 3, d_3 = 4, d_4 = 5$, and $d_5 = 8$. Since the minimum is $d_1 = 1$ and the maximum is $d_5 = 8$, the individual weights are calculated as follows:

$$w_i = \frac{d_5 - d_i}{d_5 - d_1} = \frac{8 - d_i}{8 - 1} = \frac{8 - d_i}{7}$$

This leads to the following values:

$w_1 = \frac{8-1}{7} = 1, w_2 = \frac{8-3}{7} = \frac{5}{7}, w_3 = \frac{8-4}{7} = \frac{4}{7}, w_4 = \frac{8-5}{7} = \frac{3}{7}, w_5 = \frac{8-8}{7} = 0.$

If the two nearest neighbors are positive and the remaining three are negative, then $\mathbf{x}$ is classified as positive because $\Sigma^+ = 1 + \frac{5}{7} > \Sigma^- = \frac{4}{7} + \frac{3}{7} + 0.$

Another important thing to observe is that if all nearest neighbors have the same distance from **x**, then they all get the same weight, $w_i = 1$, on account of the denominator in Eq. 3.4. The reader will easily verify that $d_k = d_1$ if and only if all the k nearest neighbors have the same distance from **x**.

3.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the motivation behind the suggestion that the vote of each of the voting neighbors should have a different weight?
- Discuss the behavior of the formula recommended in the text for the calculation of the weights.

3.6 Removing Dangerous Examples

The value of each training example can be different. Some are typical of the classes they represent, others less so, and yet others may be downright misleading. This is why it is often a good thing to pre-process the training set: to remove examples suspected of not being useful.

The method of pre-processing is guided by the two observations illustrated in Fig. 3.7. First, an example labeled with one class but surrounded by examples of another class may indicate class-label noise. Second, examples from the borderline region separating two classes are unreliable: even small amount of noise in their attribute values can shift their locations in the wrong directions, thus affecting classification. Pre-processing seeks to remove these two types of examples from the training set.

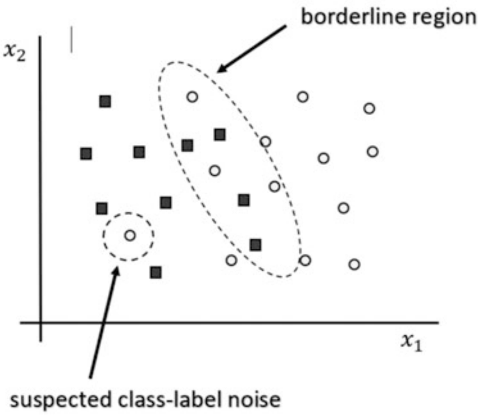


Fig. 3.7 Two potentially harmful types of examples: those surrounded by examples of a different class, and those in the "borderline region"

Table 3.5 The algorithm to identify (and remove) *Tomek Links*

```
Input: the training set of $N$ examples

1. Let $i = 1$, and let $T$ be an empty set.
2. Let $\mathbf{x}$ be the $i$-th training example, and let $\mathbf{y}$ be the nearest neighbor of $\mathbf{x}$.
3. If $\mathbf{x}$ and $\mathbf{y}$ belong to the same class, go to 5.
4. If $\mathbf{x}$ is the nearest neighbor of $\mathbf{y}$, let $T = T \cup \{\mathbf{x}, \mathbf{y}\}$.
5. Let $i = i + 1$. If $i \leq N$, go to 2.
6. Remove from the training set all examples that are now in $T$.
```

Tomek Links Before the culprit can be removed, it has to be detected. This can be accomplished by the technique of *Tomek Links*, named so after the mathematician who first mentioned a similar idea a few decades ago.[4]

Two examples, $\mathbf{x}$ and $\mathbf{y}$, are said to form a *Tomek Link* if the following three requirements are satisfied at the same time:

1. $\mathbf{x}$ is the nearest neighbor of $\mathbf{y}$.
2. $\mathbf{y}$ is the nearest neighbor of $\mathbf{x}$.
3. The class of $\mathbf{x}$ is different from the class of $\mathbf{y}$.

These conditions being typical of borderline examples and also of examples surrounded by examples of another class, it makes sense to remove from the training set all such pairs. Even this may not be enough. Sometimes, the removal of existing *Tomek Links* only creates new *Tomek Links*, and the procedure therefore has to be repeated, sometimes more than once.

The algorithm is summarized by the pseudo-code in Table 3.5, and a few instances of *Tomek Links* are shown in Fig. 3.8. Note that there are only three of them; no other pair of examples satisfies the *Tomek Link* criteria. Note, also, that the removal of the six participating examples in the picture gives rise to new *Tomek Links*.

Fewer Neighbors Are Now Needed Once the training set has been cleaned of *Tomek Links*, the number of the voting nearest neighbors can be reduced. The reason is obvious. The reader will recall that the primary motivation for high-valued k was to mitigate the negative impact of noise. Once a certain number of the noisy examples have been removed, k does not have to be so high. The experimenter may even observe that the single-neighbor 1-NN applied to the reduced set achieves the performance of a k-NN classifier applied to the original training set.

Another aspect to keep in mind is that the removal of certain examples has made the training set sparser. High values of k will then give a vote to neighbors that are no longer sufficiently similar to the examples to be classified.

[4] However, he did not use the links for machine-learning purposes.

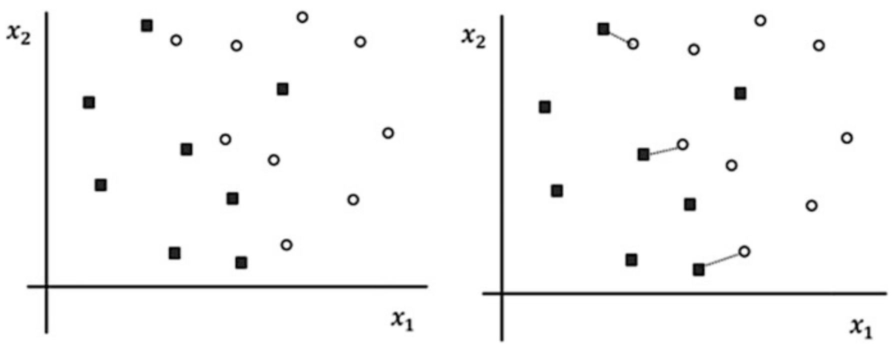


Fig. 3.8 Dotted lines connect all Tomek Links. Each participant in a Tomek Link is its partner's nearest neighbor, and each of the two examples is labeled with a different class. Removal of one set of Tomek Links can give rise to new ones

Limitation Nothing is perfect. The technique of *Tomek Links* does not identify all dangerous examples, only some of them; conversely, some of the removed examples can be "innocents" who deserved to be retained. Still, experience shows that the removal of *Tomek Links* usually does improve the overall quality of the data. The engineer has to be careful about two special situations. First, if the training set is very small, removing examples can be counterproductive. Second, when one of the classes significantly outnumbers the other, special precautions have to be made. The latter case will be discussed in Sect. 11.2.

3.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What motivates the decision to "clean" the training set? What kinds of examples do we want to remove?
- What are *Tomek Links*, and how do we identify them in the training set? Why does the procedure sometimes have to be repeated?
- How does the removal of *Tomek Links* affect the k-NN classifier? Does this removal influence the ideal setting of parameter k?

3.7 Removing Redundant Examples

Some training examples do not hurt classification, and yet we want to get rid of them because they are *redundant*: they add to computational costs without affecting the classifier's classification performance.

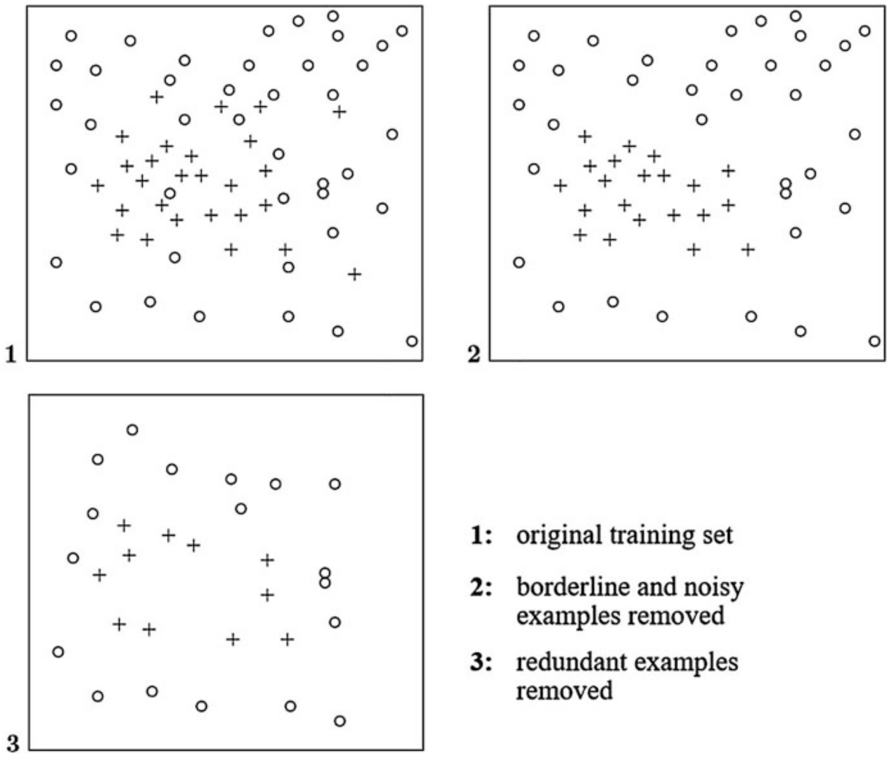


Fig. 3.9 The picture shows what happens with the training set if we remove borderline, noisy, and redundant examples

Redundant Examples and Computational Costs In machine-learning practice, we may encounter domains with 10^6 training examples described by some 10^4 attributes. Moreover, one may need to classify thousands of objects as quickly as possible. To identify the nearest neighbor of a single object, the nearest classifier relying on Euclidean distance has to carry out $10^6 \times 10^4 = 10^{10}$ arithmetic operations. Repeating this for thousands of objects results in $10^{10} \times 10^3 = 10^{13}$ arithmetic operations. This may be impractical.

Fortunately, training sets are often redundant in the sense that the k-NN classifier's behavior will be unaffected by the deletion of many training examples. Sometimes, a great majority of the examples can thus be removed with impunity. This is the case of the domain shown in the upper-left corner of Fig. 3.9.

Consistent Subset Redundancy is reduced if we replace the training set, T, with its *consistent subset*, S. In the machine-learning context, S is said to be a consistent subset of T if replacing T with S does not affect the class labels returned by the k-NN classifier. This definition, however, is not very practical because we do not know how the k-NN classifier (whether using T or S) will behave on future examples. Let

Table 3.6 Algorithm to create a consistent training subset by the removal of (some) redundant examples

```
1. Let $S$ contain one positive and one negative example, randomly selected from the training set, $T$.
2. Using examples from $S$, let $k$-NN reclassify the entire set $T$, and see which examples the
   classifier labeled correctly and which incorrectly. Let $M$ be the set of those examples that have
   been labeled incorrectly.
3. Copy to $S$ all examples from $M$.
4. If the last step did not change the contents of $S$, *stop*; otherwise, go to step 1.
```

us therefore modify the criterion: S will be regarded as a consistent subset of T if any $\mathbf{ex} \in T$ receives the same label from the classifier, no matter whether the k-NN classifier is applied to $T - \{\mathbf{ex}\}$ or to $S - \{\mathbf{ex}\}$.

Quite often, a realistic training set has many consistent subsets. How do we choose the best one? Intuitively, the smaller the subset, the better. But a perfectionist who insists on having the smallest consistent subset may come to grief because such ideal can usually be achieved only at the price of enormous computational costs. The practically minded engineer who does not believe exorbitant costs are justified will welcome a computationally efficient algorithm that "reasonably downsizes" the original set, unscientific though such formulation may appear to be.

Creating a Consistent Subset One such pragmatic technique is presented in Table 3.6. The algorithm starts by placing one random example from each class in set S. This set, S, is then used by the 1-NN classifier to decide about the labels of all training examples. At this stage, it is likely that some training examples will thus be misclassified. These misclassified examples are added to S, and the whole procedure is repeated using this larger version of S. The procedure is then repeated all over again. At a certain moment, S becomes sufficiently representative to allow the 1-NN classifier to label all training examples correctly.

3.7.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the benefit of removing redundant examples from the training set?
- What do we mean by "consistent subset"? Why is it not necessary always to look for the smallest consistent subset?
- Explain the principle of the simple algorithm that creates a reasonably sized consistent subset.

3.8 Limitations of Attribute-Vector Similarity

The successful practitioner of machine learning has to have a good understanding of the limitations of the diverse tools. Here are some ideas concerning classification based on geometric distances between attribute vectors.

Common Perception of Kangaroos Any child will tell you that a kangaroo is easily recognized by the poach on its belly. Among all the attributes describing the examples, the Boolean information about the presence or the absence of the "pocket" is the most prominent, and it is not an exaggeration to claim that its importance is greater than that of all the remaining attributes combined. Giraffe does not have it, nor does a mosquito or an earthworm.

One Limitation of Attribute Vectors Dividing attributes into relevant, irrelevant, and redundant is too crude. The "kangaroo" experience shows us that among the relevant ones, some are more important than others; a circumstance is not easily reflected in similarity measures, at least not in those discussed in this chapter.

Ideally, k-NN should perhaps weigh the relative importance of the individual attributes and adjust the similarity measures accordingly. This is rarely done, in this paradigm. In the next chapter, we will see that this requirement is more naturally addressed by linear classifiers.

Relations Between Attributes Another clearly observable feature in kangaroos is that their front legs are much shorter than the hind legs. This feature, however, is not immediately reflected by similarities derived from geometric distances between attribute vectors. Typically, examples of animals will be described by such attributes as the length of a front leg and the length of a hind leg (among many others), but relation between the different lengths is only implicit.

The reader will now agree that the classification may depend less on the original attributes than on the relations between individual attributes, such as a_1/a_2. One step further, a complex function of two or more attributes will be more informative than the individual attributes.

Low-Level Attributes In domains, the available attributes are of a very low informational level. Thus in computer vision, it is common to describe the given image by a matrix of integers, each given the intensity of one "pixel," essentially a single dot in the image. Such matrix can easily comprise millions of such pixels.

Intuitively, though, it is not these dots, very low-level attributes, but rather the way that these dots are combined into higher-level features such as lines, edges, blobs of different texture, and so on.

Higher-Level Features Are Needed The ideas presented in the last few paragraphs all converge to one important conclusion. To wit, it would be good if some more advanced machine-learning paradigm were able to create from available attributes meaningful higher-level features that would be more capable of informing us about the given object's class.

High-Level Features as a Recurring Theme in Machine Learning In later chapters, we will encounter several approaches that create meaningful higher-level features from lower-level ones. This is the case of artificial neural networks, especially those linked to the idea of *deep learning*. Also, high-level features are created by some unsupervised-learning approaches, particularly Kohonen networks and auto-encoding.

What Is *Similarity* Anyway? The reservations presented in the above paragraphs lead us to suspect that perhaps the very notion of *similarity* should somehow be induced from data. Geometric distance of attribute vectors is just one possibility. Its limitations were implied even by the opening example of this chapter, the one that posed the rhetorical question whether `giraffe` is more similar to `horse` than to `zebra`).

3.8.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Is it sufficient to divide attributes into relevant, irrelevant, and redundant? Comment on the different importance of the individual attributes.
- What is meant by the observation that some attributes are too low level and that a mechanism to create higher-level attributes is needed?

3.9 Summary and Historical Remarks

- When classifying object $\mathbf{x}$, the k-NN classifier identifies in the training set k examples most similar to $\mathbf{x}$ and then chooses the class label most common among these "nearest neighbors."
- The concrete behavior of the k-NN classifier depends to a great extent on how it evaluates similarities of attribute vectors. The simplest way to establish the similarity between $\mathbf{x}$ and $\mathbf{y}$ seems to be by calculating their geometric distance by the following formula:

$$d_M(\mathbf{x}, \mathbf{y}) = \sqrt{\Sigma_{i=1}^{n} d(x_i, y_i)} \tag{3.5}$$

  Usually, we use $d(x_i, y_i) = (x_i - y_i)^2$ for continuous-valued attributes. For discrete attributes, we put $d(x_i, y_i) = 0$ if $x_i = y_i$ and $d(x_i, y_i) = 1$ if $x_i \neq y_i$. However, more advanced methods are sometimes used.
- The use of geometric distance in machine learning can be hampered by inappropriate scales of attribute values. This is why it is usual to normalize the domains of all attributes to the unit interval, [0, 1]. The user should not forget to normalize the descriptions of future examples by the same normalization formula.

- The performance of the k-NN classifier may disappoint if many of the attributes are irrelevant. Another difficulty is presented by the diverse domains (scales) of the attribute values. The latter problem can be mitigated by normalizing the attribute values to unit intervals.
- Some examples are harmful in the sense that their presence in the training set increases error rate. Others are redundant in that they only add to computation costs without improving classification performance. Harmful and redundant examples should be removed.
- In many applications, each of the nearest neighbors has the same vote. In others, the votes are weighted by distance.
- Classical approaches to nearest-neighbor classification usually do not weigh the relative importance of individual attributes. Another limitation is caused by the fact that, in some domains, the available attributes are too detailed. A mechanism to construct from them higher-level features is then needed.

Historical Remarks The principle of the nearest-neighbor classifier was originally proposed by Fix and Hodges (1951), but the first systematic analysis was offered by Cover and Hart (1967) and Cover (1968). Exhaustive treatment of its various aspects was then provided by the book by Dasarathy (1991). The weighted k-NN classifier described here was proposed by Dudani (1975). The oldest technique to find a consistent subset of the training set was described by Hart (1968)—the one introduced in this chapter is its minor modification. The notion of *Tomek Links* is due to Tomek (1976).

3.10 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

3.10.1 Exercises

1. Determine the class of $\mathbf{y} = [4, 2]$ with the 1-NN classifier and the 3-NN classifier that are both using the training examples from Table 3.7. Explain why the classification behavior of the two classifiers differs.
2. Use the examples from Table 3.7 to classify $\mathbf{y} = [3, 3]$ with the 5-NN classifier. Note that two nearest neighbors are positive and three nearest neighbors are negative. Will weighted 5-NN classifier change anything? To see what is going on, plot the locations of the examples in a graph.
3. Again, use the training examples from Table 3.7. (a) Are there any Tomek links? (b) Can you find a consistent subset of the training set by the removal of at least one redundant training example?

Table 3.7 A simple set of training examples for the exercises

x_1	1	1	1	2	3	3	3	4	5
x_2	1	2	4	3	0	2	5	4	3
Class	+	–	–	+	+	+	–	–	–

3.10.2 Give It Some Thought

1. Discuss the possibility of applying the k-NN classifier to the "pies" domain. Give some thought to how many nearest neighbors to use, what distance metric to employ, whether to make the nearest neighbors' vote depend on distance, and so on.
2. Suggest other variations on the nearest-neighbor principle, taking inspiration from the following hints:

   (a) Introduce your own distance metrics. Do not forget that they have to satisfy the axioms mentioned at the end of Sect. 3.2.
   (b) Modify the voting scheme by assuming that some examples were created by a knowledgeable "teacher," whereas others were extracted from a database without considering how much representative each individual example may be. The teacher's examples should carry more weight.

3. Invent an alternative algorithm (different from the one in this chapter) for the removal of *redundant examples*.
4. Invent an algorithm that uses some other approach (different from the one in this chapter) to the removal of *irrelevant attributes*. Hint: withhold some training examples on which to test 1-NN classifier's performance for different subsets of attributes.

3.10.3 Computer Assignments

1. Write a program whose input is the training set, a user-specified value of k, and an object, $\mathbf{x}$. The output is the class label of $\mathbf{x}$.
2. Apply the program implemented in the previous assignment to some of the benchmark domains from the UCI repository.[5] Always take 40% of the examples out and reclassify them with the 1-NN classifier that runs on the remaining 60%.
3. Create an artificial domain consisting of 1000 examples described by a pair of attributes, each from interval [0,1]. In the square defined by these attribute values, $[0, 1] \times [0, 1]$, define a geometric figure of your own choice, and label all examples inside it as positive and all examples outside it as negative. From this initial noise-free data set, create 5 files, each obtained by changing p percent of

[5] `www.ics.uci.edu/~mlearn/MLRepository.html`.

the class labels, with $p \in \{5, 10, 15, 20, 25\}$ (thus obtaining different levels of class-label noise).

Divide each of these data files into two parts, the second to be reclassified by the k-NN classifier run the data from the first part. Observe how different values of k result in different behaviors under different levels of class-label noise.

4. Implement the Tomek Link method for the removal of harmful examples. Repeat the experiments from the previous assignments for the case where the k-NN classifier uses only examples that survived this removal. Compare the results, observing how the removal affects the classification behavior of the k-NN classifier for different values of k and for different levels of noise.

Chapter 4
Inter-Class Boundaries: Linear and Polynomial Classifiers

Each training example can be represented a point in an n-dimensional instance space. In this space, positive examples are often clustered in one region and negative examples in another. This motivates yet another machine-learning approach to classification: instead of the probabilities and similarities from the previous two chapters, the idea is to define a *decision surface* that separates the two classes. This surface can be linear—and indeed, linear functions do a good job in simple domains where examples of the two classes are easy to separate. The more flexible high-order polynomials, capable of implementing complicated shapes of inter-class boundaries have to be handled with care.

The chapter introduces two techniques for induction of *linear classifiers* from examples described by Boolean attributes and then discusses how to apply them in more general domains such as those with numeric attributes and more than two classes. The idea is then extended to *polynomial classifiers*. The powerful *support vector machines* are also mentioned.

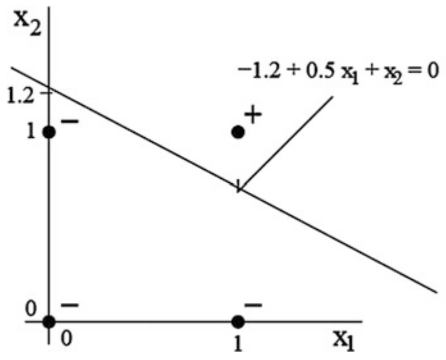
4.1 Essence

To begin, let us constrain ourselves to Boolean domains where each attribute is either *true* or *false*. To be able to use these attributes in algebraic functions, we will represent them by integers: *true* by 1, and *false* by 0.

Linear Classifier In Fig. 4.1, one example is labeled as positive and the remaining three as negative. In this particular case, the two classes are separated by the linear function defined as follows:

$$-1.2 + 0.5x_1 + x_2 = 0 \tag{4.1}$$

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_4



x_1	x_2	$-1.2+0.5x_1+x_2$	class
1	1	0.3	pos
1	0	-0.7	neg
0	1	-0.2	neg
0	0	-1.2	neg

Fig. 4.1 Linear classifier in a domain with two classes and two Boolean attributes (using 1 for *true* and 0 for *false*)

In the expression on the left-hand side, x_1 and x_2 represent attributes. If we substitute for x_1 and x_2 the concrete values of a given example (0 or 1), the expression $-1.2 + 0.5x_1 + x_2$ will be either positive or negative. The sign then determines the example's class. The table on the right shows how the four examples from the left are thus classified.

Equation 4.1 is not the only one capable of doing the job. Other expressions, say, $-1.5 + x_1 + x_2$, will label the four examples in exactly the same way. As a matter of fact, the same can be accomplished by infinitely many classifiers of the following generic form:

$$w_0 + w_1x_1 + w_2x_2 = 0$$

The function is easy to generalize to domains with n attributes:

$$w_0 + w_1x_1 + \ldots + w_nx_n = 0 \tag{4.2}$$

If $n = 2$, Eq. 4.2 defines a line; if $n = 3$, a plane; and if $n > 3$, a hyper-plane. If we introduce a "zeroth" attribute, x_0, that is not used in example description and whose value is always fixed at $x_0 = 1$, the equation can be re-written in the following compact form:

$$\sum_{i=0}^{n} w_i x_i = 0 \tag{4.3}$$

Two Practical Considerations When writing a computer program implementing the classifier, the engineer must specify how to label the rare example that finds itself exactly on the hyper-plane—which happens when the expression equals 0. Common practice either chooses the class at random or gives preference to the one that has more representatives in the training set.

Further on, let us not forget that no linear classifier can separate the positive and the negative examples if the two classes are *not* linearly separable. Thus if we change in Fig. 4.1 the class label of $\mathbf{x} = (x_1, x_2) = (0, 0)$ from minus to plus, no straight line will ever succeed. To handle situations of this kind, Sect. 4.5 will introduce polynomial classifiers. Till then, however, we will constrain ourselves to domains where the classes *are* linearly separable.

Parameters The classifier's behavior is determined by its coefficients, w_i, usually called *weights*. The task for machine learning is to find weights that allow satisfactory classification performance.

Geometrically speaking, the weights are responsible for two different tasks. Those with non-zero indexes, $w_1, \ldots w_n$, define the angle of the hyper-plane in the system of coordinates. The zeroth weight, w_0, called *bias*, determines the hyper-plane's *offset*: its distance from the origin of the system of coordinates.

Bias and Threshold In the example from Fig. 4.1, the bias was $w_0 = -1.2$. A higher value would "shift" the classifier further from the origin, $[0, 0]$, whereas $w_0 = 0$ would make the classifier pass right through the origin. Our intuitive grasp of the role played by bias in the classifier's behavior will improve if we rewrite Eq. 4.2 as follows:

$$w_1x_1 + \ldots w_nx_n = \theta \tag{4.4}$$

The term on the right-hand side, $\theta = -w_0$, is the *threshold* that the weighted sum has to exceed if the example is to be deemed positive. Note that threshold is bias with opposite sign. Thus for bias $w_0 = -1.2$, the corresponding threshold has value $\theta = 1.2$.

Simple Logical Functions Let us simplify our considerations by the somewhat extreme requirement that all attributes have the same weight, $w_i = 1$. Even under such serious constraint, careful choice of the threshold will model certain useful functions. For instance, the reader will easily verify that the following classifier returns the positive class if and only if every single attribute has $x_i = 1$, a situation known as logical AND.

$$x_1 + \ldots + x_n = n - 0.5 \tag{4.5}$$

By contrast, the next classifier returns the positive class if *at least one* attribute is $x_i = 1$, a situation known as logical OR.

$$x_1 + \ldots + x_n = 0.5 \tag{4.6}$$

Finally, the classifier below returns the positive class if *at least* k attributes (out of the total of n attributes) are $x_i = 1$. This is known as the *k-of-n* function, of which AND and OR are special cases: AND is *n-of-n*, whereas OR is *1-of-n*.

$$x_1 + \ldots + x_n = k - 0.5 \tag{4.7}$$

Weights Now that we understand the role of bias (or threshold), let us abandon the requirement that all weights be $w_i = 1$, and observe the consequences of their concrete values. For instance, consider the linear classifier defined by the following equation:

$$2 + 3x_1 - 2x_2 + 0.1x_4 - 0.5x_6 = 0 \tag{4.8}$$

The first thing to notice in the expression on the left is the absence of attributes x_3 and x_5: their zero weights, $w_3 = w_5 = 0$, render them *irrelevant* for the given classification.

As for the other attributes, their impacts depend on their weights' absolute values as well as on the signs: if $w_i > 0$, then $x_i = 1$ increases the chances of the expression being positive; and if $w_i < 0$, then $x_i = 1$ increases the chances of its being negative. Note that, in the classifier from Eq. 4.8, x_1 provides stronger support for the positive class than x_4 because $w_1 > w_4$. Likewise, the influence of x_2 is stronger than that of x_6—only in the opposite direction: by reducing the value of the overall sum, this weight makes it more likely that an example with $x_2 = 1$ will be deemed negative. Finally, the very small value of w_4 makes x_4 almost irrelevant.

Consider now the classifier defined by the following function:

$$2x_1 + x_2 + x_3 = 1.5 \tag{4.9}$$

Here the threshold 1.5 is exceeded either by the sole presence of $x_1 = 1$ (because then $2x_1 = 2 \cdot 1 > 1.5$) or by the combined contributions of $x_2 = 1$ and $x_3 = 1$. This means that x_1 will prevail when supporting the positive class even if x_2 and x_3 both support the negative class.

Low Computational Costs Note the relatively low computational costs of this approach. Whereas the 1-NN classifier had to evaluate many geometric distances, and then search for the smallest, the linear classifier only has to determine the sign of a simple expression.

4.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Write the general expression defining the linear classifier in a domain with four Boolean attributes. Why do we prefer to represent the *true* and *false* values by 1 and 0, respectively? How does the classifier determine an example's class?
- How can a linear classifier implement functions AND, OR, and *k-of-n*?
- Discuss the behavior of the linear classifier defined by $-2.5 + x_2 + 2x_3 = 0$. What do the weights tell us about the role of the individual attributes?
- Compare the computational costs of the linear classifier with those of the nearest-neighbor classifier.

4.2 Additive Rule: Perceptron Learning

Having developed some basic understanding of how the linear classifier works, we are ready to take a look at how to induce it from training data.

Learning Task Let us assume that each training example, **x**, is described by n binary attributes whose values are either $x_i = 1$ or $x_i = 0$. A positive example is indicated by $c(\mathbf{x}) = 1$, and a negative by $c(\mathbf{x}) = 0$. To make sure we do not confuse the example's real class with the one suggested by the classifier, we will denote the latter by $h(\mathbf{x})$ where the letter h emphasizes that this is the classifier's <u>h</u>ypothesis. If $\sum_{i=0}^{n} w_i x_i > 0$, the classifier "hypothesizes" that the example is positive and therefore returns $h(\mathbf{x}) = 1$. Conversely, if $\sum_{i=0}^{n} w_i x_i \leq 0$, the classifier returns $h(\mathbf{x}) = 0$. Figure 4.2 reminds us that the classifier labels **x** as positive only if the cumulative evidence supporting this class exceeds 0.

Finally, we will assume that examples with $c(\mathbf{x}) = 1$ are linearly separable from those with $c(\mathbf{x}) = 0$. This means that there exists a linear classifier that will label correctly all training examples so that $h(\mathbf{x}) = c(\mathbf{x})$ for any **x**. The task for machine learning is to find the weights, w_i, that make this happen.

Learning from Mistakes Here is the essence of the most common approach to induction of linear classifiers. Suppose we have a working version of the classifier, even if imperfect. When presented with a training example, **x**, the classifier suggests a label, $h(\mathbf{x})$. If this differs from the true class, $h(\mathbf{x}) \neq c(\mathbf{x})$, the learner concludes that the weights should be modified in a way likely to correct this error.

Let the true class be $c(\mathbf{x}) = 1$. In this event, $h(\mathbf{x}) = 0$ will only happen if $\sum_{i=0}^{n} w_i x_i < 0$, an indication that the weights are too small. If we *increase* them, the sum, $\sum_{i=0}^{n} w_i x_i$, may exceed zero, making the returned label positive, and therefore correct. Note that it is enough to increase only the weights of attributes with $x_i = 1$; when $x_i = 0$, then the value of w_i does not matter because anything multiplied by zero is still zero: $0 \cdot w_i = 0$.

Likewise, if $c(\mathbf{x}) = 0$ and $h(\mathbf{x}) = 1$, then the weights of all attributes with $x_i = 1$ should be *decreased* so as to give the sum the chance to drop below zero, $\sum_{i=0}^{n} w_i x_i < 0$, in which case the classifier will label **x** as negative.

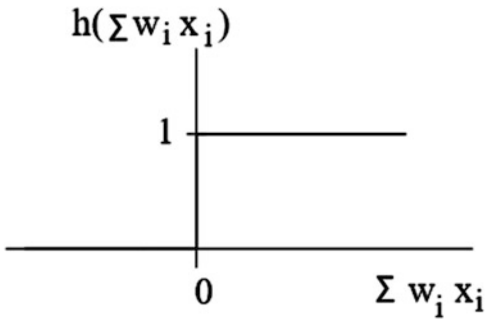


Fig. 4.2 The output of a linear classifier is $h(\mathbf{x}) = 1$ when $\sum_{i=0}^{n} w_i x_i > 0$ and $h(\mathbf{x}) = 0$ when $\sum_{i=0}^{n} w_i x_i \leq 0$, thus indicating that the example is positive or negative, respectively

Weight-Adjusting Formula The presentation of a training example, $\mathbf{x}$, can result in three different situations. The technique based on "learning from mistakes" responds to them as indicated by the following table:

Situation	Action
$c(\mathbf{x}) = 1$ while $h(\mathbf{x}) = 0$	Increase w_i for each attribute with $x_i = 1$
$c(\mathbf{x}) = 0$ while $h(\mathbf{x}) = 1$	Decrease w_i for each attribute with $x_i = 1$
$c(\mathbf{x}) = h(\mathbf{x})$	Do nothing

Interestingly, all these actions are accomplished by the same formula:

$$w_i = w_i + \eta \cdot [c(\mathbf{x}) - h(\mathbf{x})] \cdot x_i \tag{4.10}$$

Let us take a closer look at its behavior.

1. *Correct action*. If $c(\mathbf{x}) = h(\mathbf{x})$ the term in the brackets is $[c(\mathbf{x}) - h(\mathbf{x})] = 0$, which leaves w_i unchanged. If $c(\mathbf{x}) = 1$ while $h(\mathbf{x}) = 0$, the term in the brackets is 1, and the weights are increased. And if $c(\mathbf{x}) = 0$ while $h(\mathbf{x}) = 1$, the term in the brackets is negative, and the weights are reduced.
2. *Only relevant weights are affected*. If $x_i = 0$, the term to be added to the i-th weight, $\eta \cdot [c(\mathbf{x}) - h(\mathbf{x})] \cdot 0$, is zero. This means that the formula will affect the weight w_i only when $x_i = 1$.
3. *Degree of change*. How much the weight changes is controlled by the *learning rate*, η, whose user-set value is chosen from the unit interval, $\eta \in (0, 1]$,

Note that this weight modification is *additive* because a term is added to the previous weight value. Section 4.3 will introduce the alternative: *multiplicative* modification.

Perceptron Learning Algorithm Equation 4.10 is at the core of the *Perceptron Learning Algorithm*.[1] The procedure is summarized by the pseudo-code in Table 4.1. The principle is simple. Once the weights have been initialized to small random values, the training examples are presented one at a time. After each example presentation, every weight in the classifier is processed by Eq. 4.10. The last training example signals that one *training epoch* has been completed. Unless the classifier now labels correctly the entire training set, the learner returns to the first example, thus beginning the second epoch, then the third, and so on. Typically, several epochs are needed to achieve the goal.

Numeric Example Table 4.2 illustrates the procedure using a toy domain where three training examples, $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$, are described by two binary attributes. After the presentation of $\mathbf{e}_1$, the weights are reduced because $h(\mathbf{e_1}) = 1$ and $c(\mathbf{e_1}) = 0$; however, this happens only to w_0 and w_1 because $x_2 = 0$. In response to $\mathbf{e}_2$, all weights of the classifier's new version are increased because $h(\mathbf{e_2}) = 0$ and $c(\mathbf{e_2}) =$

[1]Its author, F. Rosenblatt, employed this learning technique in a device he called *Perceptron*.

Table 4.1 The *perceptron learning* algorithm

```
Assumption: the two classes, $c(\mathbf{x}) = 1$ and $c(\mathbf{x}) = 0$, are linearly separable.

1. Initialize all weights, $w_i$, to small random numbers.
   Choose an appropriate learning rate, $\eta \in (0, 1]$.
2. For each training example, $\mathbf{x} = (x_1, \dots, x_n)$, whose class is $c(\mathbf{x})$:
     i) Let $h(\mathbf{x}) = 1$ if $\sum_{i=0}^{n} w_i x_i > 0$, and $h(\mathbf{x}) = 0$ otherwise.
    ii) Update each weight by $w_i = w_i + \eta[c(\mathbf{x}) - h(\mathbf{x})] \cdot x_i$
3. If $c(\mathbf{x}) = h(\mathbf{x})$ for all training examples, stop; otherwise, return to step 2.
```

Table 4.2 Illustration of *perceptron learning*

Let the learning rate be $\eta = 0.5$, and let the randomly generated initial weights be $w_0 = 0.1$, $w_1 = 0.3$, and $w_3 = 0.4$. Set $x_0 = 1$.

Goal: Using the following training set, the perceptron learning algorithm is to learn to separate the negative examples, $\mathbf{e}_1$ and $\mathbf{e}_3$, from the positive example, $\mathbf{e}_2$.

Example	x_1	x_2	$c(\mathbf{x})$
$\mathbf{e}_1$	1	0	0
$\mathbf{e}_2$	1	1	1
$\mathbf{e}_3$	0	0	0

The linear classifier's hypothesis about $\mathbf{x}$'s class is $h(\mathbf{x}) = 1$ if $\sum_{i=0}^{n} w_i x_i > 0$ and $h(\mathbf{x}) = 0$ otherwise. After each example presentation, all weights are subjected to the same formula: $w_i = w_i + 0.5 \cdot [c(\mathbf{x}) - h(\mathbf{x})] \cdot x_i$.

The table below shows, step by step, what happens to the weights in the course of learning.

	x_1	x_2	w_0	w_1	w_2	$h(\mathbf{x})$	$c(\mathbf{x})$	$c(\mathbf{x}) - h(\mathbf{x})$
Random classifier			0.1	0.3	0.4			
Example $\mathbf{e}_1$	1	0				1	0	−1
New classifier			−0.4	−0.2	0.4			
Example $\mathbf{e}_2$	1	1				0	1	1
New classifier			0.1	0.3	0.9			
Example $\mathbf{e}_3$	0	0				1	0	−1
Final classifier			−0.4	0.3	0.9			

The final version of the classifier, $-0.4 + 0.3x_1 + 0.9x_2 = 0$, no longer misclassifies any training example. The training has thus been completed in a single epoch.

1, and all attributes have $x_i = 1$. And after $\mathbf{e}_3$, the fact that $h(\mathbf{e_3}) = 1$ and $c(\mathbf{e_1}) = 0$ results in the reduction of w_0, but not of the other weights because $x_1 = x_2 = 0$. From now on, the classifier correctly labels all training examples, and the process can thus be terminated.

Learning Depends on Initial Weights In the previous example, the training took only a single epoch, but this was only thanks to some lucky choices. First of them are the *initial weights*: different initialization may result in a different number of epochs. Most of the time, the classifier's initial version is all but useless, and a lot of training is needed before the process converges to a useful classifier. Sometimes, however, the very first version is so good that a single epoch is enough. Of course, there is also the chance, however remote, that the random-number generator creates a classifier that labels all training examples without a single error so that no training is needed.

Longer Attribute Vectors Result in Longer Training Another factor affecting the process of learning is the number of attributes that describe the training examples. As a rule of thumb, how many epochs are needed to conclude the task depends approximately linearly on the number of attributes (supposing the same learning rate, η, is used). For instance, the number of epochs needed in a domain with $3 \times n$ attributes is likely to be about three times the number of epochs needed in a domain with n attributes.

Learning Rate Critical role is played by the *learning rate*, η. In the example from Table 4.2, the reader will have noticed the rapid weight changes. For instance, w_0 jumped from 0.1 to -0.4 after the presentation of $\mathbf{e}_1$, then back to 0.1 after the presentation of $\mathbf{e}_2$, only to return to -0.4 after $\mathbf{e}_3$. Similarly strong changes were experienced also by w_1 and w_2. The phenomenon is visualized by Fig. 4.3. The reader will easily verify that the four lines represent the four successive versions of the classifier. Note how dramatic, for instance, is the change from classifier 1 to classifier 2, and then from classifier 2 to classifier 3.

This sensitivity is caused by the relatively high learning rate, $\eta = 0.5$. A smaller value, such as $\eta = 0.1$, would moderate the changes, thus "smoothing out" the learning behavior. But if we overdo it by choosing an extremely low value, say, $\eta = 0.001$, the weight changes will be minuscule, and the training process will be unnecessarily slow because a great many epochs will be needed to classify correctly all training examples.

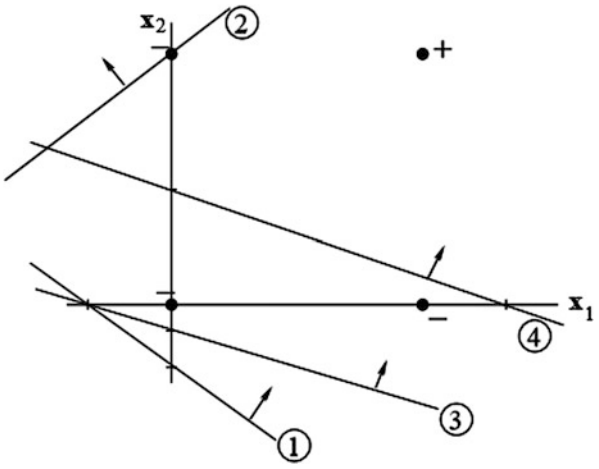


Fig. 4.3 The four classifiers from Table 4.2. The one defined by the initial weights is denoted by 1; numbers 2 and 3 represent the two intermediate stages; and 4, the final solution. The arrows indicate the half-spaces containing positive examples

If the Solution Exists, It Will Be Found Whatever the initial weights, whatever the number of binary attributes, and whatever the learning rate, one thing is *guaranteed* by a mathematical theorem: if the positive and negative classes are linearly separable, the perceptron learning algorithm will find a class-separating hyper-plane in a finite number of steps.

4.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Under what circumstances is *perceptron learning* guaranteed to find a classifier that perfectly labels all training examples?
- When does the algorithm reduce the classifier's weights, when does it increase them, and when does it leave them unchanged? Why does it modify w_i only if the corresponding attribute's value is $x_i = 1$?
- What circumstances influence the number of epochs needed by the algorithm to converge to a solution?

4.3 Multiplicative Rule: WINNOW

Perceptron learning responds to the classifier's error using an *additive rule*: a positive or negative term is added to the weights. An obvious alternative will be a *multiplicative rule* where the weights are increased or reduced as a result of being multiplied by a certain term. This is an approach adopted by WINNOW, an algorithm summarized by the pseudo-code in Table 4.3.

Table 4.3 Algorithm WINNOW

```
Assumption: the two classes, $c(\mathbf{x}) = 1$ and $c(\mathbf{x}) = 0$, are linearly separable.
Input: User-specified value of WINNOW's learning rate, $\alpha$ (usually, $\alpha \in (1, 2]$).

1. Initialize the classifier's weights to $w_i = 1$.
2. If $n$ is the number of attributes, set the threshold value to $\theta = n - 0.1$.
3. Present a training example, $\mathbf{x}$; let $c(\mathbf{x})$ be the class of $\mathbf{x}$.
   If $\sum_{i=1}^{n} w_i x_i > \theta$ then $h(\mathbf{x}) = 1$; otherwise, $h(\mathbf{x}) = 0$.
4. If $c(\mathbf{x}) \neq h(\mathbf{x})$, update the weights of each attribute whose value is $x_i = 1$:

       if $c(\mathbf{x}) = 1$ and $h(\mathbf{x}) = 0$, then $w_i = \alpha w_i$
       if $c(\mathbf{x}) = 0$ and $h(\mathbf{x}) = 1$, then $w_i = w_i / \alpha$

5. If $c(\mathbf{x}) = h(\mathbf{x})$ for all training examples, stop; otherwise, return to step 3.
```

Initialization In *perceptron learning*, the weights were initialized to small random values. In the case of WINNOW, however, they are all initially set to 1. Another difference is that *perceptron learning* requested also initialization of the weight of the artificial zeroth attribute (the one that is always $x_o = 1$), whereas WINNOW initializes to $w_i = 1$ only the weights of the real attributes (the index i acquires values from 1 to n if there are n attributes).

In place of the zeroth weight, WINNOW uses a threshold, θ. In a domain with n attributes, the threshold's value is initialized to $\theta = n - 0.1$, which is slightly less than the number of attributes.

Classification All attributes are binary, with values either 1 or 0. If the weighted some of these attributes exceeds the threshold, the classifier labels the example as positive. More specifically, the following formula is used:

$$\sum_{i=1}^{n} w_i x_i > \theta$$

If the weighted sum does not exceed the threshold, the classifier labels the example as negative.

Learning in WINNOW The general scenario is the same as in *perceptron learning*. A training example, $\mathbf{x}$, is presented, and the classifier returns a hypothesis about the example's label, $h(\mathbf{x})$. WINNOW compares this hypothesis with the known label, $c(\mathbf{x})$. If the two differ, $c(\mathbf{x}) \neq h(\mathbf{x})$, the weights of attributes with $x_i = 1$ are modified as indicated in the table below. The weights of attributes with $x_i = 0$ are left unchanged.

In the table, $\alpha > 1$ is a user-specified learning rate. Since the role of the learning rate is here somewhat different from that in *perceptron learning* (multiplicative instead of additive), a different Greek letter, α, is used.

Situation	Action
$c(\mathbf{x}) = 1$ while $h(\mathbf{x}) = 0$	$w_i = \alpha w_i$
$c(\mathbf{x}) = 0$ +while $h(\mathbf{x}) = 1$	$w_i = w_i/\alpha$
$c(\mathbf{x}) = h(\mathbf{x})$	Do nothing

The reader will easily verify that all three actions are accomplished by the following formula whose specific behavior is determined by the mutual relation between $c(\mathbf{x})$ and $h(\mathbf{x})$ in the exponent. For instance, if $c(\mathbf{x}) = h(\mathbf{x})$, the exponent is 0, and the weight does not change.

$$w_i = w_i \cdot \alpha^{c(\mathbf{x})-h(\mathbf{x})} \tag{4.11}$$

Numeric Example Table 4.4 illustrates the algorithm's behavior using a toy domain. The training set consists of all examples that can be described by three

Table 4.4 Illustration of WINNOW's behavior

Task. Using the training examples in the table on the left (below), induce the linear classifier. Let $\alpha = 2$. There being 3 attributes, the threshold is $\theta = 3 - 0.1 = 2.9$.

Weights are updated only on two occasions: presentation of $\mathbf{e}_5$ (false negative) in the first epoch and presentation of $\mathbf{e}_3$ (false positive) in the second. After these two weight modification, the classifier labels all training examples correctly.

	x_1	x_2	x_3	c(**x**)
$\mathbf{e}_1$	1	1	1	1
$\mathbf{e}_2$	1	1	0	0
$\mathbf{e}_3$	1	0	1	0
$\mathbf{e}_4$	1	0	0	0
$\mathbf{e}_5$	0	1	1	1
$\mathbf{e}_6$	0	1	0	0
$\mathbf{e}_7$	0	0	1	0
$\mathbf{e}_8$	0	0	0	0

	x_1	x_2	x_3	w_1	w_2	w_3	$h(\mathbf{x})$	$c(\mathbf{x})$
Initial class				1	1	1		
Example $\mathbf{e}_5$	0	1	1				0	1
New weights				1	2	2		
Example $\mathbf{e}_3$	1	0	1				1	0
New weights				0.5	2	1		

binary attributes. Those with $x_2 = x_3 = 1$ are labeled as positive and all others as negative, regardless of the value of irrelevant attribute x_1. The threshold is set to $\theta = 3 - 0.1 = 2.9$ because WINNOW, of course, does not know that one of the attributes is irrelevant.

When the first four examples are presented, the classifier's initial version labels them all correctly and all weights are left unchanged. The first mistake is made in the case of $\mathbf{e}_5$, a positive example for which the classifier incorrectly returns the negative label. The learner multiplies by $\alpha = 2$ the weights of attributes with $x_i = 1$ (that is, w_2 and w_3). This new classifier then classifies correctly all the remaining examples, $\mathbf{e}_6$ through $\mathbf{e}_8$. In the second epoch, the classifier errs on $\mathbf{e}_3$, a negative example misclassified as positive. In response to this error, the algorithm reduces weights w_1 and w_3 (but not w_2 because $x_2 = 0$). After this last weight modification, the classifier labels correctly the entire training set.

Note that the weight of the irrelevant attribute, x_1, is now smaller than the weights of the relevant attributes. Indeed, the ability to penalize irrelevant attributes by significantly reducing their weights, thus "winnowing them out," hence the name, is one of the main advantages of this technique over *perceptron learning*.

Parameter α Parameter α controls the learner's sensitivity to errors in a manner reminiscent of the learning rate in *perceptron learning*. One important difference is the requirement that $\alpha > 1$. This guarantees an increase in w_i in the case of a false negative, and a decrease in w_i in the case of a false positive.

The parameter's concrete value is not completely arbitrary. If it exceeds 1 by just a little (say, if $\alpha = 1.1$), the weight updates are very small, and convergence is slow (requiring too many epochs). Increasing α's value accelerates convergence but risks overshooting the solution. The ideal value is best established experimentally. Good results are often achieved with $\alpha = 1.5$ but, of course, each domain is different.

No Negative Weights? Let us point out another fundamental difference between WINNOW and *perceptron learning*. Since the (originally positive) weights are always multiplied by α or $1/\alpha$, none of them can ever drop to zero, let alone turn negative. This means that unless appropriate measures are taken a whole class of linear classifiers has thus been eliminated: those with negative or zero coefficients.

The absence of negative weights is removed if we replace each of the original attributes by a pair of "new" attributes: one copying the original attribute's value, the other having the opposite value (1 instead of 0 or 0 instead of 1). In a domain that originally had n attributes, the total number of attributes will thus be $2n$, the value of the $(n+i)$th attribute, x_{n+i}, being the opposite of x_i.

For instance, suppose that an example is described by the following three attribute values:

$$x_1 = 1, x_2 = 0, x_3 = 1$$

In the new representation, the same example will be described by six attributes:

$$x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 0$$

For these, WINNOW will have to find six weights, $w_1, \ldots, w_6$, or perhaps seven, if w_0 is used.

WINNOW Versus Perceptron In comparison with *perceptron learning*, WINNOW tends to converge faster in domains with irrelevant attributes whose weights are quickly reduced to small values. However, neither of the two, WINNOW nor *perceptron learning*, will recognize (and eliminate) *redundant* attributes. For instance, in the event of two attributes always having the same value, $x_i = x_j$, the learning process will converge to the same weight for both, making them look equally important even though it is clear that only one of them is needed.

4.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What formula is used by the weight-updating mechanism in WINNOW? Why is the formula called *multiplicative*?
- What is the shortcoming of multiplying or dividing the weights by $\alpha > 1$? How is the situation remedied?
- Summarize the differences between WINNOW and *perceptron learning*. What is WINNOW's main advantage?

4.4 Domains with More Than Two Classes

Having only two sides, a hyper-plane may separate the positive examples from the negative examples—and that is all. When it comes to *multi-class domains*, the tool seems helpless. Or is it?

Groups of Binary Classifiers What exceeds the powers of an individual can be solved by a team. One practical solution is shown in Fig. 4.4. The "team" consists of four binary classifiers, each specializing on one of the four classes, C_1 through C_4. Ideally, the presentation of an example from C_i results in the i-th classifier returning $h_i(\mathbf{x}) = 1$, and all the other classifiers returning $h_j(\mathbf{x}) = 0$, assuming, again, that each class is linearly separable from the other classes.

Modifying the Training Data To exhibit this behavior, the individual classifiers need to be properly trained. This training can be accomplished by any of the two algorithms from the previous sections. The only additional trick is that the engineer needs to modify the training data.

Table 4.5 illustrates the principle. On the left is the original training set, T, where each example is labeled with one of the four classes. On the right are four "derived" sets, T_1 through T_4, each consisting of the same six examples which have now been re-labeled so that an example that in the original set, T, represents class C_i is labeled with $c(\mathbf{x}) = 1$ in T_i and with $c(\mathbf{x}) = 0$ in all other sets.

Needing a Master Classifier The training sets, T_i, are presented to a program that induces from each of them a linear classifier dedicated to the corresponding class. This is not the end of the story, though. The training examples may poorly represent the classes, they may be corrupted by noise, and even the requirement of linear separability may be violated. As a result, the induced classifiers may overlap each other in the sense that two or more of them will respond to the same example, $\mathbf{x}$, with $h_i(\mathbf{x}) = 1$, leaving the incorrect impression that $\mathbf{x}$ simultaneously belongs to

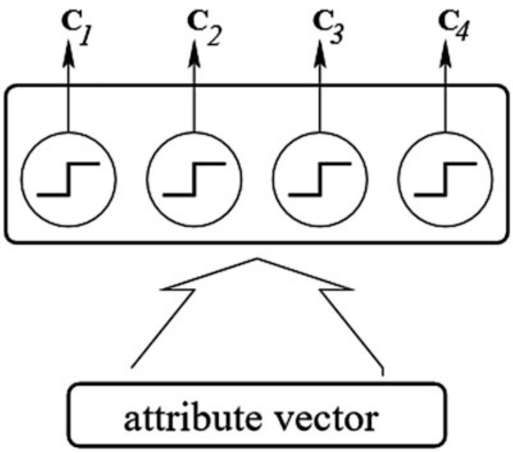


Fig. 4.4 Converting a 4-class problem into four 2-class problems

Table 4.5 A 4-class training set, T, converted to 4 binary training sets, $T_1 \dots T_4$

T	
e_1	C_2
e_2	C_1
e_3	C_3
e_4	C_4
e_5	C_2
e_6	C_4

T_1	
e_1	0
e_2	1
e_3	0
e_4	0
e_5	0
e_6	0

T_2	
e_1	1
e_2	0
e_3	0
e_4	0
e_5	1
e_6	0

T_3	
e_1	0
e_2	0
e_3	1
e_4	0
e_5	0
e_6	0

T_4	
e_1	0
e_2	0
e_3	0
e_4	1
e_5	0
e_6	1

more than one class. This is why a *master classifier* is needed; its task is to choose from the returned classes the one most likely to be correct.

This is not difficult. The reader will recall that a linear classifier labels $\mathbf{x}$ as positive if the weighted sum of $\mathbf{x}$'s attribute values exceeds zero, $\Sigma_{i=0}^{n} w_i x_i > 0$. This sum (usually different in each of the classifiers that have returned $h_i(\mathbf{x}) = 1$) can be interpreted as the amount of evidence in support of the corresponding class. The master classifier then simply gives preference to the class whose binary classifier delivered the highest $\Sigma_{i=0}^{n} w_i x_i$.

Numeric Example The principle is illustrated by Table 4.6 where each row represents a different class (with the total of four classes). Each classifier has a different set of weights, each weight represented by one column in the table. When an example is presented, its attribute values are in each classifier multiplied by the corresponding weights. When checking the correctness of these calculations, do not forget that $x_4 = 0$, so that w_4 is always ignored. We observe that in two classifiers, C_2 and C_3, the weighted sums are positive, $\Sigma_{i=0}^{n} w_i x_i > 0$, which means that these two classifiers both return $h(\mathbf{x}) = 1$. Given that each example is to be labeled with one and only one class, the master classifier has to make a choice. In this particular case, it gives preference to C_2 because this classifier's weighted sum is greater than that of C_3.

Practical Limitations A little disclaimer is in place here. This method of employing linear classifiers in multi-class domains is reliable only if the number of classes is moderate, say, 3–5. In domains with many classes, the "derived" training sets, T_i, will be imbalanced in the sense that most examples will have $c(\mathbf{x}) = 0$ and only a few $c(\mathbf{x}) = 1$. As we will learn in Sect. 11.2, imbalanced training sets tend to cause difficulties in noisy domains unless appropriate measures have been taken.

Also, do not forget the requirement that each class has to be linearly separable from the others.

Table 4.6 Illustration of the master classifier's task: to choose the example's class from two or more candidates

Suppose we have four binary classifiers (the i-th classifier is used for the i-th class) defined by the weights listed in the table below. How shall the master classifier label example $\mathbf{x} = (x_1, x_2, x_3, x_4) = (1, 1, 1, 0)$?

Class	Classifier					$\Sigma_{i=0}^{n} w_i x_i$	$h(\mathbf{x})$
	w_0	w_1	w_2	w_3	w_4		
C_1	−1.5	1	0.5	−1	−5	−1	0
C_2	0.5	1.5	−1	3	1	4	1
C_3	1	−2	4	−1	0.5	2	1
C_4	−2	1	1	−3	−1	−3	0

The right-most column tells us that two classifiers, C_2 and C_3, return $h(\mathbf{x}) = 1$. From these, C_2 is supported by the higher value of $\Sigma_{i=0}^{n} w_i x_i$. Therefore, the master classifier labels $\mathbf{x}$ with C_2.

4.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- When trying to use N linear classifiers in an N-class domain, how will you create the training sets, T_i, for the induction of the individual binary classifiers?
- How is the example's class chosen in a situation where two or more binary classifiers return $h(\mathbf{x}) = 1$?

4.5 Polynomial Classifiers

Let us now abandon the strict requirement that positive examples be *linearly* separable from negative ones. Quite often, they are not. Not only can the linear separability be destroyed by noise; the very shape of the region occupied by one of the classes can render linear decision surface inadequate. Thus in the training set shown in Fig. 4.5, no linear classifier ever succeeds in separating the two squares from the circles. Such separation can only be accomplished by a non-linear curve such as the parabola shown in the picture.

Non-linear Classifiers The point having been made, we have to ask *how* to induce these *non-linear classifiers* from data. To begin with, we have to decide what type of function to employ. This is not difficult. Math teaches us that *any* n-dimensional curve can be approximated to arbitrary precision with some *polynomial* of a sufficiently high order. Let us therefore take a look at how to induce from data these polynomials. Later, we will discuss their practical utility.

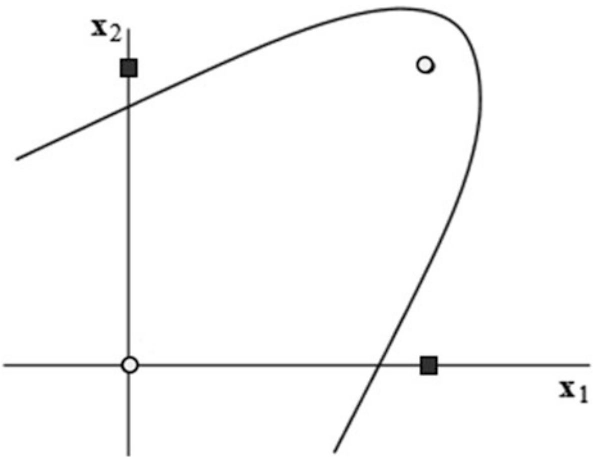


Fig. 4.5 In some domains, no linear classifier can separate the positive examples from the negative. Only a *non-linear classifier* can do so

Polynomials of the Second Order The good news is that the coefficients of polynomials can be induced by the same techniques that we have used for linear classifiers. Let us explain how.

For the sake of clarity, we will begin by constraining ourselves to simple domains with only two Boolean attributes, x_1 and x_2. The second-order polynomial is then defined as follows:

$$w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2 = 0 \tag{4.12}$$

The expression on the left is a sum of terms that have one thing in common: a weight, w_i, multiplies a product $x_1^k x_2^l$. In the first term, we have $k + l = 0$, because $w_0x_1^0x_2^0 = w_0$; next come the terms with $k + l = 1$, concretely, $w_1x_1^1x_2^0 = w_1x_1$ and $w_2x_1^0x_2^1 = w_1x_2$; and the sequence ends with three terms that have $k + l = 2$: specifically, $w_3x_1^2$, $w_4x_1^1x_2^1$, and $w_5x_2^2$. The thing to remember is that the expansion of the second-order polynomial stops when the sum of the exponents reaches 2.

Of course, some of the weights can be $w_i = 0$, rendering the corresponding terms "invisible" such as in $7 + 2x_1x_2 + 3x_2^2$ where the coefficients of x_1, x_2, and x_1^2 are zero.

Polynomials of the r-th Order More generally, the r-th order polynomial in a two-dimensional domain is a sum of weighted terms, $w_ix_1^kx_2^l$, such that $k + l = j$, where $j = 0, 1, \ldots r$.

The reader will easily make the next step and write down the general formula that defines the r-th order polynomial for domains with more than two attributes. A hint: the sum of the exponents in any single term never exceeds r.

Converting Polynomials to Linear Classifiers Whatever the polynomial's order, and whatever the number of attributes, the task for machine learning is to find the weights that result in the separation of positive examples from negative examples.

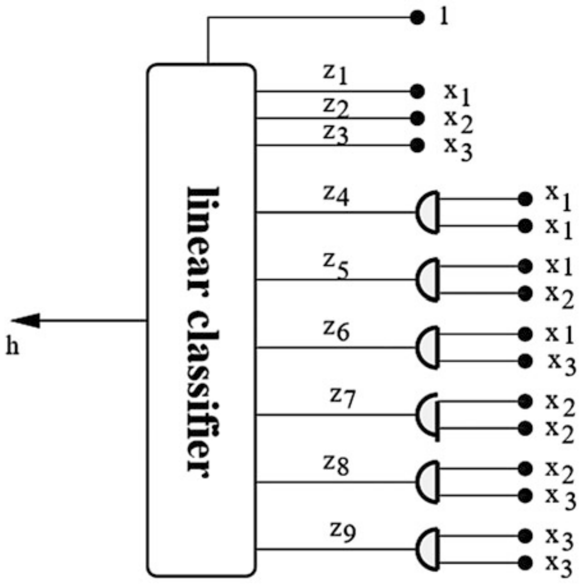


Fig. 4.6 A polynomial classifier can be converted into a linear classifier with the help of multipliers that pre-process the data

The seemingly unfortunate circumstance that the terms are non-linear (the sum of the exponents may exceed 1) is easily removed by *multipliers*. Thus in a domain with binary inputs, the multiplier outputs a logical conjunction of inputs: 1 if *all* inputs are 1; and 0 if *at least one* input is 0. With the help of multipliers, each product of attributes can be replaced by a new attribute, z_i. Equation 4.12 is then re-written as follows:

$$w_0 + w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 = 0 \tag{4.13}$$

Note, for instance, that $z_3 = x_1^2$ and $z_4 = x_1 \cdot x_2$. This "trick" has transformed the originally non-linear problem with two attributes, x_1 and x_2, into a linear problem with five (newly created) attributes, z_1 through z_5.

Figure 4.6 illustrates the situation where a second-order polynomial is used in a domain with three attributes.

Induction of Linear Classifier Can Be Used Since the values of z_i in each example are known, the weights can be obtained without any difficulties by *perceptron learning* or by WINNOW. Of course, we must not forget that these techniques will find the solution only if the polynomial of the chosen order is indeed capable of separating the two classes.

For simplicity, we have limited our discussions to Boolean attributes. In Sect. 4.7, we will learn that if the weights are trained by the perceptron learning algorithm, we usually succeed even if the attributes are numeric. WINNOW, however, should preferably be used with Boolean attributes. Mechanisms to employ this technique in numeric domains do exist, but their detailed discussion exceeds the scope of an introductory text. Suffice it to say that one possibility is to "binarize" the continuous-valued attributes by the technique used in numeric decision trees (see Sect. 5.4).

4.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- When do we resort to non-linear classifiers? What argument speaks in favor of polynomial classifiers?
- Write down the mathematical expression that defines a polynomial classifier. Explain the meaning of "r-th order."
- What "trick" allows us to train polynomial classifiers with the same techniques that are used for linear classifiers?

4.6 Specific Aspects of Polynomial Classifiers

Now that we understand that the main strength of polynomials is their almost unlimited flexibility, it is time to turn our attention to their shortcomings and limitations.

Overfitting Polynomial classifiers tend to *overfit* noisy training data. Since the problem of overfitting is typical of many machine-learning paradigms, it is a good idea discuss its essence in some detail. Let us constrain ourselves to two-dimensional continuous domains that are easy to visualize.

The eight training examples in Fig. 4.7 fall into two groups. In one group, all examples are positive (empty circles); in the other, all save one are negative (filled circles). Two attempts at separating the two classes are shown. The one on the left uses a linear classifier, ignoring the fact that one training example is thus misclassified. The one on the right resorts to a polynomial classifier in an attempt to avoid any error on the training set.

Inevitable Trade-Off Which of the two is to be preferred? The answer is not straightforward because we do not know the underlying nature of the data. It may be that the two classes are linearly separable, and the only cause for one positive example to be found in the negative region is class-label noise. If this is the case, the single error made by the linear classifier on the training set is inconsequential, whereas the polynomial on the right, cutting deep into the negative area, will misclassify those future examples that find themselves on the wrong side of the curve. Conversely, it is possible that the outlier *does* represent some legitimate, even if rare, aspect of the positive class. In this event, the use of the polynomial is justified. Practically speaking, however, the assumption that the single outlier is only noise is more likely to be correct than the "special-aspect" alternative.

A realistic training set will contain not one, but quite a few, perhaps many examples that appear to be in the wrong area of the instance space. And the inter-class boundary that the classifier seeks to approximate may indeed be curved, though *how much* curved is anybody's guess. The engineer may regard the linear classifier as too crude, and opt instead for the more flexible polynomial. This said, a high-order polynomial will separate the two classes even in a very noisy training set—and then fail miserably on future data. The ideal solution is usually somewhere between the extremes and has to be determined experimentally.

How Many Weights? The total number of the weights to be trained depends on the the number of attributes, and on the polynomial's order. A simple analysis would reveal that, in the case of n attributes and the r-th order polynomial, the number of weights is determined by the following expression known from combinatorics:

$$N_W = \binom{n+r}{r} \tag{4.14}$$

Note that N_W is impractically high for high values of n. For instance, even for the relatively modest case of $n = 100$ attributes and a polynomial's order $r = 3$, the number of weights to be trained is $N_W = 176{,}851$ (103 choose 3). For today's computers, the computational costs thus incurred are not insurmountable. More serious is the danger of overfitting the noisy training set; the polynomial is simply too flexible to be trusted. The next paragraphs will tell us *how much* flexible.

Capacity of Linear Classifiers The trivial domain in Fig. 4.1 consisted of four examples. Given that each example can be labeled as either positive or negative, we have $2^4 = 16$ different ways of assigning positive and negative labels to them. Of

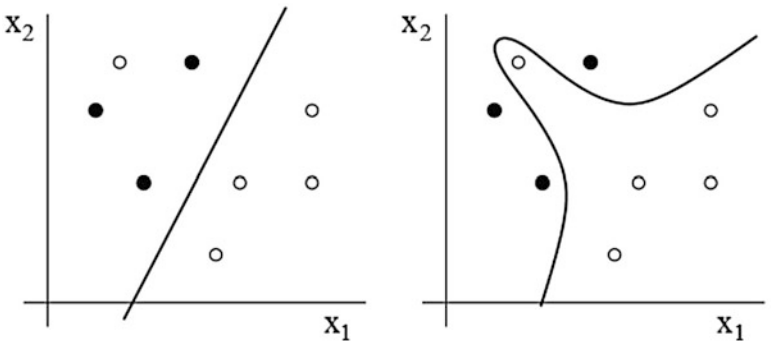


Fig. 4.7 The two classes are linearly separable, but noise has caused one example to be mislabeled. The polynomial on the right *over-fits* the data

these sixteen ways, only in two cases are the two classes not linearly separable. In other words, linear inseparability is here a rare event. But what will the situation look like in the more general case where m examples are described by n attributes? What are the chances that a random labeling of the examples will result in linearly separable classes?

Mathematics has found a simple guideline to be used in domains where n is "reasonably high" (say, ten or more attributes): if the number of examples, m, is less than twice the number of attributes ($m < 2n$), the probability that a random distribution of the two labels among the examples will result in linear separability is close to one hundred percent. Conversely, this probability converges to zero when $m > 2n$. This is why we say that "the *capacity* of a linear classifier is twice the number of attributes."

Capacity of Polynomial Classifiers The result from the previous paragraph applies to polynomial classifiers, too. The role of attributes is here played by the terms, z_i, obtained by the multipliers. Their number, N_W, is given by Eq. 4.14. We have seen that N_W can be quite high—which makes the capacity high, too. In the case of $n = 100$ and $r = 3$, the number of weights is 176,851. This means that the 3rd-order polynomial is almost certain to separate perfectly the two classes if the size of the training set is less than 353,702; and it will do so regardless of any noise in the data.

For this reason, polynomials are prone to overfit noisy training sets.

4.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is meant by *overfitting*? Explain why overfitting is difficult to avoid in polynomial classifiers.
- What is the upper bound on the number of weights to be trained in a polynomial of the r-th order in a domain that has n attributes?
- What is the *capacity* of the linear or polynomial classifier? What does capacity tell us about linear separability?

4.7 Support Vector Machines

Now that we understand that polynomial classifiers do not call for any new learning algorithms, we can return to linear classifiers, a topic we have not yet exhausted. Let us abandon the restriction to the Boolean attributes, and consider also the possibility of the attributes being continuous. Can we then still rely on the two training algorithms described above?

Perceptron Learning in Numeric Domains In the case of *perceptron learning*, the answer is easy: yes, the same weight-modification formula can be used. Practical experience shows, however, that it is a good idea to normalize all attribute values so that they fall into the unit interval, $x_i \in [0, 1]$. We can use to this end the normalization technique described in the chapter dealing with nearest-neighbor classifiers, in Sect. 3.3.

Let us repeat, for the reader's convenience, the weight-adjusting formula:

$$w_i = w_i + \eta[c(\mathbf{x}) - h(\mathbf{x})]x_i \tag{4.15}$$

Learning rate, η, and the difference between the real and hypothesized class labels, $[c(\mathbf{x})-h(\mathbf{x})]$, have the same meaning and impact as before. What has changed is the role of x_i. In the case of Boolean attributes, the value of $x_i = 1$ or $x_i = 0$ decided whether or not the corresponding weight should change. Here, however, the value of x_i decides *how much* the weight should be affected: the change is greater if the attribute's value is higher.

Multiplicative Rule Also when working with WINNOW, we can essentially use the same learning formula as in the case of Boolean attributes:

$$w_i = w_i \alpha^{c(\mathbf{x})-h(\mathbf{x})} \tag{4.16}$$

However, we have to be careful about *when* to apply the formula. Previously, WINNOW modified only the weights of attributes with $x_i = 1$. Now that the attribute values come from a continuous domain, some modifications are needed. One possibility is the following rule:

Update weight w_i only if the value of the i-th attribute is $x_i \geq 0.5$.

Classes That Are Not Linearly Separable Let us only remark that both algorithms, *perceptron learning* and WINNOW, usually find a relatively low-error solution even if the two classes are *not* linearly separable, say, in the presence of noise. Such success is not guaranteed by any theorem, only by practical experience.

Which Linear Classifier Is Best? Another question needs attention. Figure 4.8 shows three linear classifiers, each perfectly separating positive training examples from negative ones. Knowing that "good behavior" on the training set does not guarantee high performance in the future, we have to ask: which of the three is likely to score best on future examples?

Support Vector Machine (SVM) In Fig. 4.8, the dotted-line classifier all but touches the nearest examples on either side; we say that this classifier has a small *margin*. The margin is much greater in the case of the solid-line classifier: the nearest examples on either side are much more distant from the line than in the case of the other classifiers. Mathematicians have been able to prove that the greater the margin, the higher the chances that the classifier will do well on future data. This theoretical result has very practical consequences.

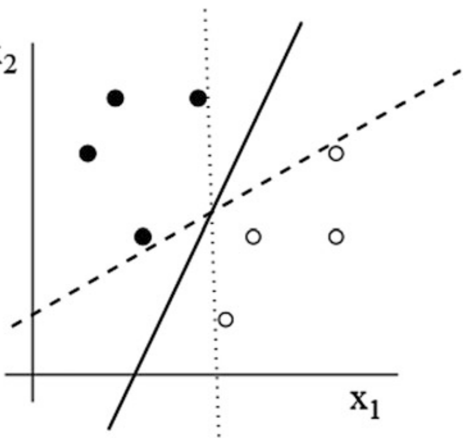


Fig. 4.8 Which of the classifiers that work perfectly on the training set will do best on future data?

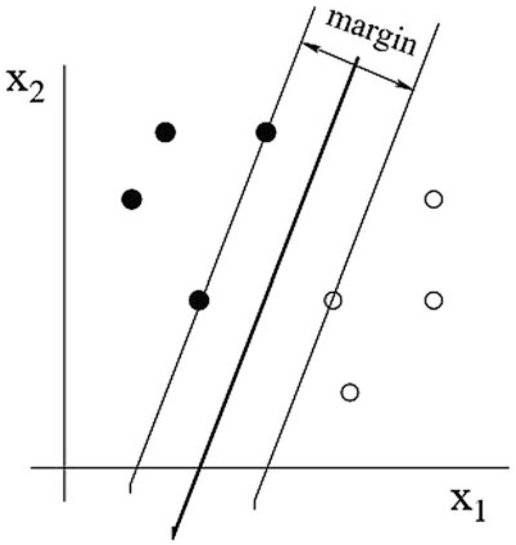


Fig. 4.9 The technique of the *support vector machines* looks for a separating hyper-plane with the maximum *margin*

Figure 4.9 presents the principle of the so-called *support vector machines, (SVM)*. The solid line represents the best classifier, the one that maximizes the margin, the graph shows also two thinner lines, parallel to the classifier, each at the same distance from it. They are here to visualize the margin. The reader can see that they pass through the examples nearest to the classifier. These examples are called *support vectors* (recall that each example is a vector of attributes), thus giving the name to the approach. Note that the classifier is located right in the middle between the nearest example on either side.

The task for machine learning is to identify the support vectors that maximize the margin. The simplest technique may try all possible n-tuples of examples and measure the margin implied by each such choice. This, of course, is hardly practical in domains with big training sets.

Algorithms that find the optimum support vectors in an efficient manner are rather advanced and rely on some fancy mathematics; their treatment would surely be beyond the ambitions of an introductory text. In reality, not many people would consider implementing such an algorithm on their own. In the past, engineers relied on software packages available for free on the internet. More recently, popular programming languages have built-in SVM-functions, saving the programmer's time and effort.

4.7.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Can *perceptron learning* and WINNOW be used in numeric domains? Are any modifications needed?
- Given that there are infinitely many linear classifiers capable of separating the positive examples from the negative (assuming such separation exists), which of them can be expected to give the best results on future data?
- What is a *support vector*? What do we have in mind when we say that the *margin* is to be maximized?
- How would you summarize the principle of a support vector machine, SVM? Is it necessary to know how to create it?

4.8 Summary and Historical Remarks

- Linear and polynomial classifiers define a *decision surface* that separates the positive examples from the negative examples. Specifically, *linear* classifiers label the examples according to the sign of the following expression where x_i is the value of the i-the attribute and w_0 is the classifier's *bias*:

  $$w_0 + w_1 x_1 + \ldots w_n x_n$$

  The concrete behavior is determined by the weights, w_i. The task for machine learning is to find weights that maximize classification performance.
- Typical induction techniques rely on "learning from mistakes." Training examples are presented to the learner one at a time. Whenever the learner misclassifies an example, the weights are modified in a way likely to reduce the error. When the entire training set has been presented, one *epoch* has been completed. Usually, several epochs are needed.

- Two weight-modification techniques were considered here: the additive rule of *perceptron learning*, and the multiplicative rule of WINNOW.
- In domains with more than two classes, one can induce a specialized classifier for each class. A *master classifier* then chooses the class whose classifier had the highest value of $\Sigma_{i=0}^{n} w_i x_i$.
- In domains with non-linear class boundaries, *polynomial* classifiers can sometimes be used. A second-order polynomial in a two-dimensional domain is defined by the following expression:

$$w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2$$

- The weights of the polynomial can be found by the same learning algorithms as in the case of linear classifiers, provided that the non-linear terms (e.g., x_1x_2) have been replaced (with the help of *multipliers*) by newly created attributes such as, for example, $z_3 = x_1^2$ or $z_4 = x_1x_2$.
- Polynomial classifiers are prone to *overfit* noisy training data. This is mainly due to the high flexibility resulting from the very high number of trainable weights.
- The potentially best class-separating hyper-plane (among the infinitely many candidates) is identified by the technique of the *support vector machines (SVM)*. The idea is to maximize the distance of the nearest positive and the nearest negative example from the classifier's hyper-plane.

Historical Remarks The principle of *perceptron learning* was developed by Rosenblatt (1958), whereas WINNOW was proposed and analyzed by Littlestone (1987). The question of the capacity of linear and polynomial classifiers was analyzed by Cover (1965). The principle of Support Vector Machines was invented by Vapnik (1995) as one of the consequences of the Computational Learning Theory which will be discussed in Chap. 7.

4.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

4.9.1 Exercises

1. Write down equations for linear classifiers to implement the following functions:

   - At least two out of the Boolean attributes $x_1, \ldots, x_5$ are *true*

- At least three out of the Boolean attributes $x_1, \ldots, x_6$ are *true*, and at least one of them is *false*.

2. Return to the examples from Table 4.2. Hand-simulate the perceptron learning algorithm's procedure, starting from a different initial set of weights than the one used in the table. Try also a different learning rate.
3. Repeat the same exercise, this time using WINNOW. Do not forget to introduce the additional "attributes" for what in perceptrons were the negative weights.
4. Write down the equation that defines a third-order polynomial in two dimensions. How many multipliers (each with up to three inputs) would be needed if we wanted to train the weights using the perceptron learning algorithm?

4.9.2 Give It Some Thought

1. How can induction of linear classifiers be used to identify irrelevant attributes? Hint: try to run the learning algorithm on different subsets of the attributes, and then observe the error rate achieved after a fixed number of epochs. Another hint: look at the weights.
2. Explain in what way it is true that the 1-NN classifier applied to a pair of examples (one positive, the other negative) in a plane defines a linear classifier. Invent a machine-learning algorithm that uses this observation in yet another way of creating linear classifiers. Generalize the procedure to n-dimensional domains.
3. Under what circumstances is a linear classifier likely to have better classification performance on independent testing examples than a polynomial classifier?
4. Sometimes, a linearly separable domain becomes linearly non-separable on account of class-label noise. Think of a technique capable of removing such noisy examples. Hint: you may rely on an idea from the chapter on k-NN classifiers.

4.9.3 Computer Assignments

1. Implement the perceptron learning algorithm and run it on the following training set where six examples (three positive and three negative) are described by four attributes:

   Observe that the linear classifier fails to reach zero error rate because the two classes are not linearly separable.
2. Create a training set consisting of 20 examples described by five binary attributes, $x_1, \ldots, x_5$. Examples in which at least three attributes have values $x_i = 1$ are labeled as positive, all other examples are labeled as negative. Using this training set as input, induce by perceptron learning a linear classifier. Experiment with different values of learning rate, η. Plot a function where the horizontal axis represents η, and the vertical axis represents the number of example-

x_1	x_2	x_3	x_4	Class
1	1	1	0	**pos**
0	0	0	0	**pos**
1	1	0	1	**pos**
1	1	0	0	**neg**
0	1	0	1	**neg**
0	0	0	1	**neg**

presentations needed for the classifier to correctly classify all training examples. Discuss the results.

3. Use the same domain as in the previous assignment (five Boolean attributes, and the same definition of the positive class). Add to each example N additional Boolean attributes whose values are determined by a random-number generator. Vary N from 1 through 20. Observe how the number of example-presentations needed to achieve the zero error rate depends on N.
4. Again, use the same domain, but add attribute noise by changing the attribute values in randomly selected examples (while leaving class labels unchanged). Observe what minimum error rate can then be achieved.
5. Repeat the last three assignments for different sizes of the training set, evaluating the results on (always the same) testing set of examples that were not used in learning.
6. Design an experiment showing that the performance of K binary classifiers, connected in parallel as in Fig. 4.4, will decrease if we increase the number of classes. How much will this observation by affected by noise?
7. Run induction of linear classifiers on selected Boolean domains from the UCI repository[2] and compare the results.
8. Experimenting with selected domains from the UCI repository, observe the impact of learning rate, η, on the convergence speed of the perceptron learning algorithm.
9. Compare the behaviors of linear and polynomial classifiers. Observe how the former wins in simple domains, and the latter in highly non-linear domains.

[2]www.ics.uci.edu/~mlearn/MLRepository.html.

Chapter 5
Decision Trees

The classifiers discussed in previous chapters expect that all attribute values be known at the same time; classification is then based on the complete attribute vector that describes the example. In some applications, this scenario is unrealistic. A physician looking for the cause of her patient's ailment may have nothing to begin with save a few subjective symptoms. To narrow the field of possible diagnoses, she prescribes a few lab tests, and, based on their results, additional lab tests still. In other words, the doctor considers only "attributes" likely to add to her momentary understanding and ignores the remaining attributes.

Here is the lesson. Quite often, exhaustive information about attribute values may not be immediately available—and it may not even be needed, provided that the classifier focuses on one attribute at a time, always choosing the attribute that offers maximum relevant information. A popular tool built around these thoughts is known as a *decision tree*.

The chapter shows how to use decision trees to classify examples and introduces a simple technique that induces them from data. The reader will learn how to benefit from *pruning*, how to convert the tree to *if-then* rules, and how to use the tree for data analysis.

5.1 Decision Trees as Classifiers

The training set in Table 5.1 consists of eight examples described by three attributes, and labeled as positive or negative instances of a given class. For convenience, let us assume, for the time being, that all attributes are discrete. Later, when the underlying principles become clear, we will generalize the approach for domains with continuous attributes or with mixed continuous and discrete attributes.

Decision Tree Figure 5.1 shows a few example decision trees that correctly classify the data from Table 5.1. Internal nodes contain attribute-value tests, the edges

Table 5.1 Eight training examples described by three discrete attributes and classified as positive and negative examples of a given class

Example	crust size	shape	filling size	Class
*e*1	big	circle	small	**pos**
*e*2	small	circle	small	**pos**
*e*3	big	square	small	**neg**
*e*4	big	triangle	small	**neg**
*e*5	big	square	big	**pos**
*e*6	small	square	small	**neg**
*e*7	small	square	big	**pos**
*e*8	big	circle	big	**pos**

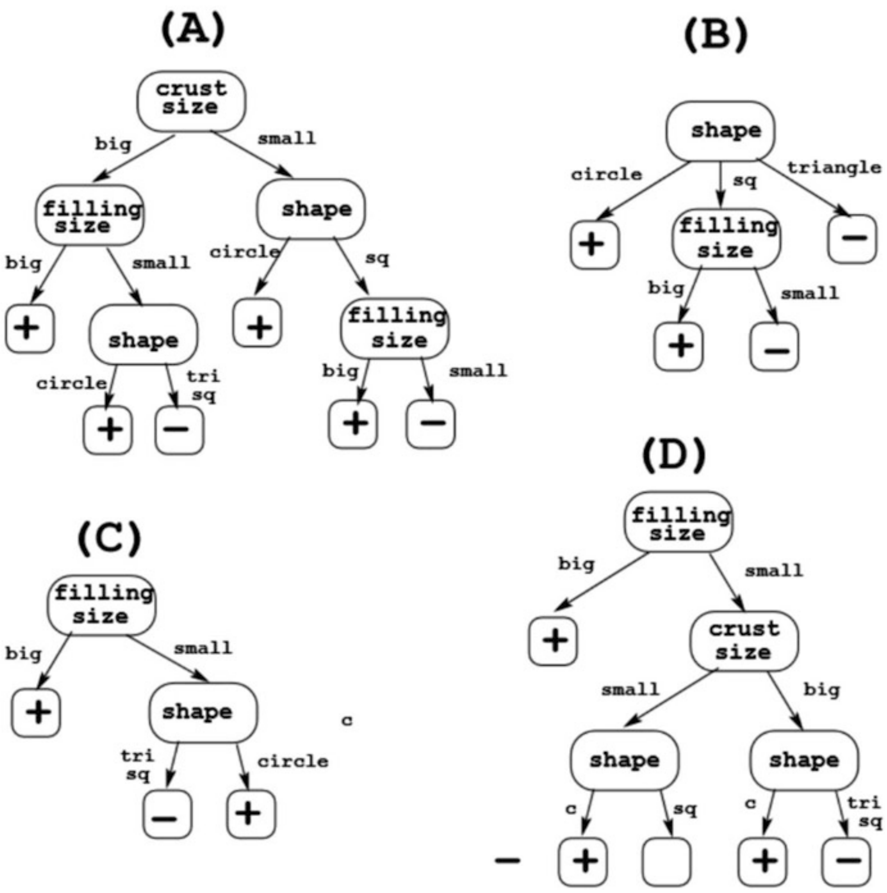


Fig. 5.1 Example decision trees for the "pies" domain. Note how each tree differs in size and in the way the tests are ordered. Each tree classifies correctly all training examples from Table 5.1

indicate how to proceed based on the test results, and the leafs[1] represent class labels. An example to be classified is first subjected to the test prescribed at the topmost node, the *root*. The test's outcome then decides which edge leading from the root to follow. The process continues until a leaf node is reached, and the example is then labeled with the class associated with this leaf.

Let us illustrate the process using the tree from Fig. 5.1b. The root asks about `shape`, and each of the three edges starting at the root represents one possible outcome. In examples $e1$, $e2$, and $e8$, the `shape` is `circle`, and the examples are therefore sent down the left edge—which ends in a leaf with the label `pos`; this indeed is the class that Table 5.1 gives for all these three examples. In $e4$, the `shape` is `triangle`, and the corresponding edge ends in a leaf labeled `neg`, again the correct class. Somewhat more complicated is the situation with examples $e3$, $e5$, $e6$, and $e7$ where the `shape` is `square`. The corresponding edge does not end in a leaf, but rather in an internal node that asks for the value of `filling-size`. In the case of $e5$ and $e7$, the value is `big`, which leads to a leaf labeled `pos`. In the other two examples, $e3$ and $e6$, the value is `small`, which sends them to a leaf labeled `neg`.

We have convinced ourselves that the decision tree from Fig. 5.1b identifies the correct class for all training examples. By way of a little exercise, the reader may want to verify that the other three trees in the picture are just as successful.[2]

Supporting Explanations Comparing this classifier with those discussed in the previous chapters, we notice one advantage: *interpretability*. If anybody asks why example $e1$ is positive, the answer is, "because its `shape` is `circle`." The other paradigms do not offer explanations. Bayesian and linear classifiers are typical *black boxes*: when presented with an example, they simply return its class and never offer any reasons. The situation is only slightly better in the case of the *k-NN* classifier that does offer a semblance of a rudimentary argument: for instance, "$\mathbf{x}$ is `pos` because this is the class of the training example most similar to $\mathbf{x}$." This is a far cry from the explicit attribute-based explanation offered by a decision tree.

One can go one step further and interpret a decision tree as a set of rules such as "if `shape=square` and `filling-size=big`, then choose class `pos`." A domain expert inspecting these rules may tell us whether they make sense, and whether they agree with his or her view of the given domain. For instance, the filling may be black when poppy is used and white in the case of cheese, and inspection of a concrete rule can lead to the conclusion that Johnny prefers poppy to cheese. In this sense, induction of decision trees can generate useful new *knowledge*.

The expert may also be able to point out spurious tests that have found their way into the data structure only due to some statistical glitch, an apparent regularity that—just like the constellations in the night skies—cannot be traced to any concrete material cause. A data analysis and an expert can then join forces in an effort to eliminate the spurious tests "manually," thus further improving the decision tree.

[1]In technical literature, both spellings are used: *leaves* and *leafs*.

[2]Note that the decision tree can serve as a simple mechanism for data compression.

Missing Edges The reader will recall that, in linear classifiers, an example could find itself exactly on the class-separating hyper-plane, in which case the class was selected more or less at random. Something similar may happen in decision trees, too. Suppose the tree from Fig. 5.1a is used to determine the class of the following example:

```
(crust-size=small) AND (shape=triangle) AND (filling-size=small)
```

Let us follow the procedure step by step. The root asks about `crust-size`. The value being `small`, the classifier sends the example down the right edge, to the test on `shape`. Here, only two outcomes seem possible: `circle` or `square`, but not `triangle`. The reason is that when the tree was being created, it was not known that an object with `crust-size=small` could be triangular. Nothing of that kind appeared in the training set, and there thus did not seem to be any reason to create the corresponding edge. Even if the edge were created, it would not be clear where it should point to.

The engineer implementing this classifier in a computer program must make sure the program is instructed what to do in the case of "missing edges." Choosing the class at random or preferring the most frequent class are the most obvious possibilities but of course, a concrete application may call for different measures.

5.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Describe the mechanism that uses a decision tree to classify examples. Illustrate the procedure using the decision trees from Fig. 5.1 and the training examples from Table 5.1.
- What do we mean by the statement that, "the decision tree's choice of a concrete class can be explained"? Is something similar possible in the case of other classifiers?
- Under what circumstances can a decision tree be unable to determine an example's class? How would you handle the situation as a programmer?

5.2 Induction of Decision Trees

Let us first suggest a very crude induction algorithm. Applying it to a training set, we will realize that a great variety of alternative decision trees can thus be obtained. A brief discussion will then convince us that, among these, the smaller ones are better. This observation will motivate an improved version of the technique.

Divide and Conquer Let us try to create a decision tree manually. Suppose we decide that the root node should test the value of `shape`. In the training set, three different outcomes are found: `circle, triangle`, and `square`. For each, the classifier will need a separate edge leading from the root. The first, defined by `shape=circle`, will be followed by examples $T_C = \{e1, e2, e8\}$; the second, defined by `shape=triangle`, will be followed by $T_T = \{e4\}$; and the last, defined by `shape=square`, will be followed by $T_S = \{e3, e5, e6, e7\}$. Each of the three edges will lead from the root to another node, either an attribute test or a leaf.

Seeing that all examples in T_C are positive, we will let this edge end in a leaf labeled with `pos`. Similarly, the edge followed by the examples from T_T will end in a leaf labeled with `neg`. Similar decision, however, is impossible in the case of the last edge because T_S contains both classes. To be able to reach a class-containing leaf, we need to place at the end of this edge another test, say, `filling-size`. This attribute can acquire two values, `small` and `big`, dividing T_S into two subsets. Of these, $T_{S-S} = \{e3, e6\}$ is characterized by `filling-size=small`; the other, $T_{S-B} = \{e5, e7\}$, is characterized by `filling-size=big`. Seeing that all examples in T_{S-S} are positive, and all examples in T_{S-B} are negative, we let both edges end in leafs, the former labeled with `pos`, the latter with `neg`. At this stage, the tree-building process can stop because each training example is guaranteed to reach a leaf.

The reader will remember that each tree node is associated with a set of examples that pass through it or end in it. Starting with the root, each test divides the training set into disjoint subsets which are then divided into further subsets, and so on, until each subset is “pure” in the sense that all its examples belong to the same class. The approach is known as the *divide-and-conquer* technique.

Alternative Trees In the process just described, the arbitrary choice of `shape` and `filling-size` as attribute tests results in the decision tree in Fig. 5.1b. To get used to the mechanism, the student is encouraged to experiment with alternative choices such as placing at the root the tests on `crust-size` or `filling-size`, and considering different options for the tests at the lower level(s). Quite a few other decision trees will thus be created, some of them shown in Fig. 5.1.

That so many solutions can be obtained even in this very simple domain is a reason for concern. Is there a way to decide which trees to prefer? If so, we can hope that an improved version of the divide-and-conquer technique will create a “good” tree *by design*, and not by mere chance.

Tree Size The smallest of the data structures in Fig. 5.1 contains two attribute tests; the largest, five. We will see that differences may have a strong impact on the classifier’s behavior. Before proceeding to the various aspects of this statement, however, let us emphasize that the number of nodes in the tree is not the only way to measure its size; just as important is the number of tests that have to be carried out when classifying an average example.

Thus in a domain where `shape` is almost always either `circle` or `triangle` (and only very rarely `square`), the average number of tests prescribed by the tree from Fig. 5.1b will only slightly exceed 1 because both `shape=circle` and `shape=triangle` immediately lead to leafs with concrete class labels. But if the prevailing `shape` is `square`, the average number of tests approaches 2. Quite often, then, a bigger tree may result in fewer tests than a smaller three. The engineer needs to have a clear mind as to what matters: minimum number of tests applied to an average example, or minimum number of nodes found in the decision tree.

Small Trees Versus Big Trees There are several reasons why small decision trees (those with few tests) are preferred. One of them is *interpretability*. Human expert find it easy to analyze, explain, and perhaps even correct, a decision tree that consists of no more than a few tests. The larger the tree, the more difficult this is.

Another advantage of small decision trees is their tendency to dispose of *irrelevant* and *redundant* information. Whereas the relatively large tree from Fig. 5.1a employs all three attributes, the smaller one from Fig. 5.1b is just as good at classifying the training set—without ever considering `crust-size`. Such economy will come handy in domains where certain attribute values are expensive or time-consuming to obtain.

Finally, larger trees are prone to *overfit* the training set. This is because the divide-and-conquer method keeps splitting the training set into smaller and smaller subsets, the number of these splits being the number of attribute tests in the tree. Ultimately, the resulting training subsets can become so small that the classes may get separated by an attribute that only by chance—or noise—has a different value among the remaining positive and negative examples.

Induction of Small Decision Trees When illustrating the behavior of the divide-and-conquer technique with manual tree-building, we picked the attributes at random, and then observed that some choices resulted in smaller trees. Apparently, attributes differ in the amount of information they convey. For instance, `shape` is capable of immediately labeling some examples as positive (if the value is `circle`) or negative (if the value is `triangle`); but `crust-size` cannot do so unless assisted by another attribute.

Assuming that there is a way to measure the amount of information offered by each attribute (one such mechanism will be explained in Sect. 5.3), we are ready to formalize the technique for induction of decision trees by a pseudo-code—see Table 5.2.

Table 5.2 Recursive procedure to induce a decision tree

```
Let $T$ be the training set.
grow($T$):

(1) Find the attribute, $at$, that contributes the maximum information about the class labels.
(2) Divide $T$ into subsets, $T_i$, each characterized by a different value of $at$.
(3) For each $T_i$:
    If all examples in $T_i$ belong to the same class, create a leaf labeled with this class; otherwise,
    apply the same procedure recursively to each training subset: grow($T_i$).
```

5.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of the divide-and-conquer technique for induction of decision trees.
- What are the advantages of small decision trees as compared to large ones?
- What determines the size of the decision tree obtained by the divide-and-conquer technique?

5.3 How Much Information in an Attribute?

To create a compact decision tree, the divide-and-conquer technique relies on one critical component: the ability to decide how much information about the class labels is conveyed by the individual attributes. This section introduces a mechanism to calculate this quantity.

Information Contents of a Message Suppose the training examples are labeled as pos or neg, the relative frequencies of these two classes in the training set being $p_{\mathtt{pos}}$ and $p_{\mathtt{neg}}$.[3] Let us select a random training example. How much information is conveyed by the message, "this example's class is pos"?

The answer depends on $p_{\mathtt{pos}}$. In the extreme case, where all examples are known to be positive, $p_{\mathtt{pos}} = 1$, the message does not tell us anything new. We know the example is positive without being told so; the amount of information in this message is zero. The situation is different when both classes are equally represented, $p_{\mathtt{pos}} = p_{\mathtt{neg}} = 0.5$. Here, our guess is no better than flipping a coin, and a message about the class label *does* provide some information. And if a great majority of examples are known to be negative so that, say, $p_{\mathtt{pos}} = 0.01$, we are all but certain that the selected example is negative. The message that this is not the case is unexpected, which means that its information contents are high. We see that the lower the value of $p_{\mathtt{pos}}$, the more information the message conveys.

When quantifying the information contents of a message of this kind, the following formula has been found convenient:

$$I_{\mathtt{pos}} = -\log_2 p_{\mathtt{pos}} \tag{5.1}$$

The negative sign compensates for the fact that the logarithm of $p_{\mathtt{pos}} \in (0, 1)$ is always negative. Table 5.3 shows the information contents for some typical values of $p_{\mathtt{pos}}$. Note that the unit for the amount of information is 1 *bit*. Another comment:

[3]Recall that relative frequency of pos is the percentage of examples labeled with pos. We use it to estimate the probability that a randomly picked example will be positive.

Table 5.3 Some values of the information contents of the message, "this randomly drawn example is positive." Note that the message is impossible for $p_{\mathrm{pos}} = 0$

p_{pos}	$-\log_2 p_{\mathrm{pos}}$
1.00	0 bits
0.50	1 bit
0.25	2 bits
0.125	3 bits

the base of the logarithm being always 2, it common to write $\log p_{\mathrm{pos}}$ instead of the more rigorous $\log_2 p_{\mathrm{pos}}$.

Entropy: Average Information Contents So much for the information contents of a single message. Suppose, now, that someone picks one training example at a time, always telling us what its class label is, until all training examples have been checked. Both messages will occur, "the example is positive" and "the example is negative," the first with probability p_{pos}, the second with probability p_{neg}. To calculate the average information contents, we weigh the information contents of each of the two messages by their probabilities (or relative frequencies) in the training set, T:

$$H(T) = -p_{\mathrm{pos}} \log_2 p_{\mathrm{pos}} - p_{\mathrm{neg}} \log_2 p_{\mathrm{neg}} \tag{5.2}$$

There seems to be a problem, here: the logarithm of zero is not defined, and Eq. 5.2 thus appears meaningless if $p_{\mathrm{pos}} = 0$ or $p_{\mathrm{neg}} = 0$. Fortunately, a simple analysis (using limits and the l'Hopital rule) will reveal that, for p converging to zero, the expression $p \log p$ converges to zero, too, and we conclude that $0 \cdot \log 0 = 0$.

$H(T)$ is called *entropy* of T. Its value reaches its maximum, $H(T) = 1$, when $p_{\mathrm{pos}} = p_{\mathrm{neg}} = 0.5$ (because $0.5 \cdot \log 0.5 + 0.5 \cdot \log 0.5 = 1$); and it drops to its minimum, $H(T) = 0$, when either $p_{\mathrm{pos}} = 1$ or $p_{\mathrm{neg}} = 1$ (because $0 \cdot \log 0 + 1 \cdot \log 1 = 0$).

The case with $p_{\mathrm{pos}} = 1$ or $p_{\mathrm{neg}} = 1$, which results in $H(T) = 0$, is regarded as perfect regularity because all examples in the set belong to the same class. Conversely, the case with $p_{\mathrm{pos}} = p_{\mathrm{neg}} = 0.5$, which results in $H(T) = 1$, is seen as a total lack of any regularity. This is why entropy is sometimes said to quantify the amount of "chaos" in the data.

Amount of Information Contributed by an Attribute The concept of entropy (lack of regularity) will help us confront an important question: how much does the knowledge of a discrete attribute's value tell us about an example's class?

Let us remind ourselves that the attribute, *at*, divides the training set, T, into subsets, T_i, each characterized by a different value of *at*. Quite naturally, each subset will be marked by its own probabilities (relative frequencies) of the two classes, p_{ipos} and p_{ineg}. Using these values and Eq. 5.2, we get the entropy, $H(T_i)$, of each subset.

Let $|T_i|$ be the number of examples in T_i, and let $|T|$ be the number of examples in the whole training set, T. The probability that a randomly drawn training example belongs to T_i is estimated as follows:

$$P_i = \frac{|T_i|}{|T|} \tag{5.3}$$

With this, we are ready to calculate the weighted average of all the entropies:

$$H(T, at) = \Sigma_i P_i \cdot H(T_i) \tag{5.4}$$

The result, $H(T, at)$, is the entropy of a system where not only the class label but also the value of attribute *at* are known for each training example. The amount of information contributed by *at* is the difference between the system's entropy *without at* being considered and the system's entropy *with* this attribute being considered:

$$I(T, at) = H(T) - H(T, at) \tag{5.5}$$

It would be easy to prove that this difference cannot be negative; information can only be gained, never lost, by considering *at*. In certain rare cases, however, $I(T, at) = 0$, which means that *at* does not contribute any information about the training examples' class labels.

Applying Eq. 5.5 separately to each attribute, we can establish which of them provides the maximum amount of information, and as such is the best choice for the "root" test in the first step of the algorithm from Table 5.2.

The best-attribute-choosing procedure is summarized by the pseudo-code in Table 5.4. The process starts by the calculation of the entropy of the system where only class percentages are known. Next, the algorithm calculates the information gain offered by each attribute. The attribute that leads to the highest information gain is best.

Table 5.4 The algorithm to find the most informational attribute

```
1. Calculate the entropy of the training set, $T$, using the percentages, $p_{\rm pos}$ and $p_{\rm neg}$, of the
   positive and negative examples:

        $H(T) = -p_{\rm pos} \log_2 p_{\rm pos} - p_{\rm neg} \log_2 p_{\rm neg}$

2. For each attribute, $at$, that divides $T$ into subsets, $T_i$, with relative sizes $P_i$, do the following:

     (i) for each subset, $T_i$, calculate its entropy and denote it by $H(T_i)$;
    (ii) calculate the system's average entropy: $H(T, at) = \Sigma_i P_i \cdot H(T_i)$;
   (iii) calculate information gain: $I(T, at) = H(T) - H(T, at)$

3. Choose the attribute with the highest information gain.
```

Numeric Example Table 5.5 shows how to select the most informative attribute in the domain from Table 5.1. At the beginning, the entropy, $H(T)$, of the system without attributes is established. Next, we observe that `shape` divides the training set into three subsets. The average of their entropies, $H(T, \texttt{shape})$, is calculated, and the difference between $H(T)$ and $H(T, \texttt{shape})$ gives the information conveyed by this attribute. Repeating the procedure for `crust-size` and `filling-size`

Table 5.5 Illustration of the search for the attribute with maximum information

Example	`crust` size	`shape`	`filling` size	Class
$e1$	`big`	`circle`	`small`	**pos**
$e2$	`small`	`circle`	`small`	**pos**
$e3$	`big`	`square`	`small`	**neg**
$e4$	`big`	`triangle`	`small`	**neg**
$e5$	`big`	`square`	`big`	**pos**
$e6$	`small`	`square`	`small`	**neg**
$e7$	`small`	`square`	`big`	**pos**
$e8$	`big`	`circle`	`big`	**pos**

Here is the entropy of the training set where only class labels are known:

$$\begin{aligned} H(T) &= -p_{\text{pos}} \log_2 p_{\text{pos}} - p_{\text{neg}} \log_2 p_{\text{neg}} \\ &= -(5/8)\log(5/8) - (3/8)\log(3/8) \quad = 0.954 \end{aligned}$$

Next, we calculate the entropies of the subsets defined by the values of `shape`:

$$\begin{aligned} H(\texttt{shape} = \texttt{square}) &= -(2/4)\cdot\log(2/4) - (2/4)\cdot\log(2/4) = 1 \\ H(\texttt{shape} = \texttt{circle}) &= -(3/3)\cdot\log(3/3) - (0/3)\cdot\log(0/3) = 0 \\ H(\texttt{shape} = \texttt{triangle}) &= -(0/1)\cdot\log(0/1) - (1/1)\cdot\log(1/1) = 0 \end{aligned}$$

Now we obtain average entropy of a system with known class labels *and* `shape`:

$$H(T, \texttt{shape}) = (4/8)\cdot 1 + (3/8)\cdot 0 + (1/8)\cdot 0 = 0.5$$

Repeating the same procedure for the other two attributes, we get the following:

$$\begin{aligned} H(T, \texttt{crustsize}) &= 0.951 \\ H(T, \texttt{crustsize}) &= 0.607 \end{aligned}$$

The values below give the information gains for all attributes:

$$\begin{aligned} I(T, \texttt{shape}) &= H(T) - H(T, \texttt{shape}) &&= 0.954 - 0.5 &&= 0.454 \\ I(T, \texttt{crustsize}) &= H(T) - H(T, \texttt{crustsize}) &&= 0.954 - 0.951 &&= 0.003 \\ I(T, \texttt{fillingsize}) &= H(T) - H(T, \texttt{fillingsize}) &&= 0.954 - 0.607 &&= 0.347 \end{aligned}$$

Maximum information is contributed by `shape`.

and comparing the results, we realize that `shape` contributes more information than the other attributes. We thus choose `shape` for the root test.

This, by the way, is how the decision tree from Fig. 5.1b was obtained.

A Comment on Logarithms The information-contents formulas from the previous paragraphs operate with logarithms base 2. The reader will recall how they are calculated from natural logarithms:

$$\log_2 x = \frac{\ln x}{\ln 2}$$

Since the difference is only the constant coefficient $\ln 2$, the same best attribute is identified whether $\log_2 x$ or $\ln x$ is used. The only difference is that, when natural logarithms are used, the amount information is no longer given in *bits*.

5.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What do we mean when we talk about the "amount of information conveyed by a message"? How is this amount determined, and what units are used?
- What is *entropy* and how does it relate to the frequency of the positive and negative examples in the training set?
- How do we use entropy when assessing the amount of information contributed by a given attribute?

5.4 Binary Split of a Numeric Attribute

The entropy-based mechanism from the previous section expected all attributes to be discrete. With a small modification, however, the same approach can be applied also to continuous attributes. All we need is to convert them to Boolean attributes.

Converting a Continuous Attribute to a Boolean Attribute Let us denote the continuous attribute by x. The idea is to choose a threshold, θ, and then decide that a newly created Boolean attribute will be *true* if $x < \theta$ means and *false* if $x \geq \theta$ (or the other way round).

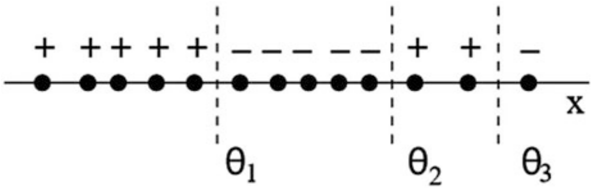
How to establish the best possible value of θ? Here is one possibility. Suppose that x has a different value in each of the N training examples. Let us sort these values in ascending order, denoting by x_1 the smallest, and by x_N the highest. Any pair of neighboring values, x_i and x_{i+1}, then defines a threshold, placed right in the middle between them: $\theta_i = (x_i + x_{i+1})/2$. For instance, a four-example training set where x has values 3, 4, 6, and 9 leads us to consider $\theta_1 = 3.5, \theta_2 = 5.0$, and $\theta_3 = 7.5$. For each of these $N - 1$ thresholds, we will calculate the amount of information offered by the Boolean attribute thus defined and then choose the threshold where this information gain is maximized.

Candidate Thresholds The approach just described incurs high computational costs. Indeed, in a domain with one hundred thousand examples described by one hundred attributes (which is nothing extraordinary), the information contents of $10^5 \times 10^2 = 10^7$ different thresholds has to be calculated. This seems unrealistic; fortunately, mathematicians have proved that a great majority of these thresholds can be ignored so that the costs are reduced to a mere fraction.

Table 5.6 shows the principle of this reduction. In the upper part, thirteen values of x are ordered from left to right, each labeled with the class (positive or negative) of the training example in which the value was found. Here is the rule: the best threshold is never found between values with the same class label. Put another way,

Table 5.6 Illustration of the search for the best threshold

The values of attribute x are sorted from the smallest to the highest. The candidate thresholds are those located between values labeled with opposite class labels.



Here is the entropy of the training set, ignoring attribute values:

$H(T) = -p_+ \log p_+ - p_- \log p_-$

$\quad = -(7/13)\log(7/13) - (6/13)\log(6/13) = 0.9957$

Here are the entropies of the training subsets defined by the 3 candidate thresholds:

$H(x < \theta_1) = -(5/5)\log(5/5) - (0/5)\log(0/5) = 0$

$H(x > \theta_1) = -(2/8)\log(2/8) - (6/8)\log(6/8) = 0.8113$

$H(x < \theta_2) = -(5/10)\log(5/10) - (5/10)\log(5/10) = 1$

$H(x > \theta_2) = -(2/3)\log(2/3) - (1/3)\log(1/3) = 0.9183$

$H(x < \theta_3) = -(7/12)\log(7/12) - (5/12)\log(5/12) = 0.9799$

$H(x > \theta_3) = -(0/1)\log(0/1) - (1/1)\log(1/1) = 0$

Average entropies associated with the individual thresholds:

$H(T, \theta_1) = (5/13).0 + (8/13).0.8113 = 0.4993$

$H(T, \theta_2) = (10/13).1 + (3/13).0.9183 = 0.9811$

$H(T, \theta_3) = (12/13).0.9799 + (1/13).0 = 0.9045$

Information gains entailed by the individual candidate thresholds:

$I(T, \theta_1) = H(T) - H(T, \theta_1) = 0.9957 - 0.4993 = 0.4964$

$I(T, \theta_2) = H(T) - H(T, \theta_2) = 0.9957 - 0.9811 = 0.0146$

$I(T, \theta_3) = H(T) - H(T, \theta_3) = 0.9957 - 0.9045 = 0.0912$

Threshold θ_1 gives the highest information gain.

Table 5.7 Algorithm to find the best numeric-attribute test

```
1. For each attribute $at_i$:
    (i) Sort the training examples by the values of $at_i$;
    (ii) Determine the candidate thresholds, $\theta_{ij}$, as those lying between examples with opposite labels;
    (iii) For each $\theta_{ij}$, determine the amount of information contributed by the Boolean attribute thus created.
2. Choose the pair $[at_i, \theta_{ij}]$ with the highest information gain.
```

it is enough to calculate the information gain only for locations between neighboring examples with different classes. In the specific case shown in Table 5.6, only three *candidate thresholds*, θ_1, θ_2, and θ_3, need to be investigated (among the three, θ_1 is shown to be best).

The Root of a Numeric Decision Tree The algorithm summarized by the pseudocode in Table 5.7 determines the best attribute test for the root of a decision tree in a domain where all attributes are continuous. Note that the test consists of a pair, $[at_i, \theta_{ij}]$, where at_i is the selected attribute and θ_{ij} is the best threshold found for this attribute. If an example's value of the i-th attribute is below the threshold, $at_i < \theta_{ij}$, the left branch of the decision tree is followed; otherwise, the right branch is chosen.

5.4.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the idea behind the intention to divide the domain of a continuous attribute into two parts.
- What mathematical finding helps us reduce the apparently prohibitive computational costs?

5.5 Pruning

Section 5.2 mentioned some virtues of small decision trees: interpretability, removal of irrelevant and redundant attributes, lower danger of overfitting. These were the arguments that motivated the use of information gain in decision tree induction; they also motivate the step that usually follows: *pruning*.

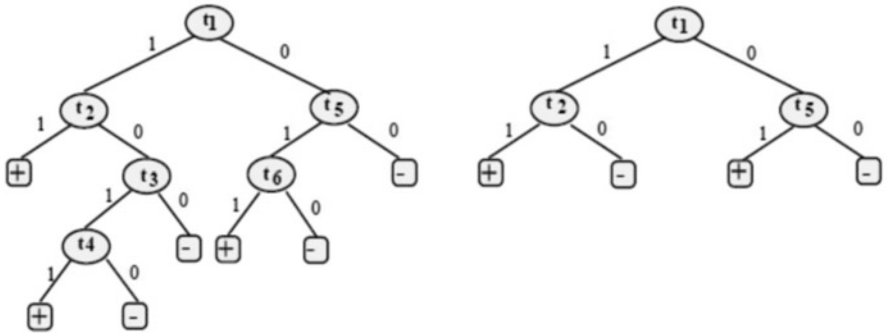


Fig. 5.2 A simple approach to pruning will replace a subtree with a leaf. In the case depicted here, two such subtrees were replaced

The Essence Figure 5.2 illustrates the principle. On the left is the original decision tree with six tests: $t_1, \dots t_6$. On the right is its pruned version. Closer inspection reveals that the subtree rooted in test t_3 in the original tree has in the pruned tree been replaced with a leaf labeled with the negative class; and the subtree rooted in test t_6 has been replaced with a leaf labeled with the positive class. Here is the point: pruning replaces one or more subtrees with leafs, each labeled with the class most common among the training examples that in the original classifier reach the removed subtree.

The idea may sound counter-intuitive: induction sought a tree with zero errors on the training examples, but this perfection may be lost in the pruned tree! There is no need to be alarmed, though. The ultimate goal is *not* to classify the training examples (their classes are known anyway). We want to label *future* examples, and pruning tends to improve the tree's performance on those future examples.

Error Estimate Pruning is typically carried out in a sequence of steps. Replace with a leaf one subtree, then another, and so on, as long as the replacements appear to be beneficial by a reasonable criterion. The term "beneficial" is meant to warn us that we do not want to pay for the smaller-tree advantages with compromised classification performance.

The last consideration brings us to the issue of *error estimate*. Let us return to Fig. 5.2. Let m be the number of training examples that reach test t_3 in the decision tree on the left. If we replace the subtree rooted in t_3 by a leaf (as in the tree on the right), some of these m examples may become misclassified. Denoting the number of these misclassified examples by e, we can estimate the probability of an example's misclassification (at this leaf) by relative frequency: e/m. However, knowing that small values of m will render such estimate unreliable, we prefer the following formula where N is the total number of training examples:

$$E_{estimate} = \frac{e+1}{N+m} \tag{5.6}$$

The reader should recall (or re-read) what Sect. 2.3 had to say about the m-estimate of the probabilities of rare events.

Error Estimates for the Whole Tree Let us again return to Fig. 5.2. The tree on the left has two main subtrees, one rooted at t_2, the other at t_5. Let m_2 and m_5 be the numbers of the training examples reaching t_2 and t_5, respectively; and let E_2 and E_5 be the error estimates (obtained by Eq. 5.6) of the two subtrees. For the total of $N = m_2 + m_5$ training examples, the error rate of the whole subtree is estimated as the weighted average of the two subtrees:

$$E_R = \frac{m_2}{N} E_2 + \frac{m_5}{N} E_5 \tag{5.7}$$

Of course, in a situation with more than just two subtrees, the weighted average has to be taken over all of them. This should present no major difficulties.

As for the values of E_2 and E_5, these are obtained from the error rates of the specific subtrees, and these again from the error rates of their sub-subtrees, and so on, all the way down to the lowest-level tests. The error-estimating procedure is thus recursive in principle.

Suppose that the subtree to be pruned is the one rooted at t_3, which happens to be one of the two children of t_2. The error estimate for t_2 is calculated as the weighted average of E_3 and the error estimate for the other child of t_2 (the leaf labeled with the positive class). The resulting estimate would then be combined with E_5 as shown above.

Post-Pruning The term *post-pruning* refers to the scenario where the decision tree was supposed to be pruned *after* it has been fully induced (an alternative will be discussed in the next subsection). The essence is to replace a subtree with a leaf labeled with the class most frequent among the training examples reaching that leaf. Since there are usually several (or many) subtrees that can thus be replaced, a choice has to be made; and the existence of a choice calls for a criterion to guide the decision.

Here is one possibility. We know that pruning is likely to affect the classifier's performance. One way of predicting *how much* the performance is going to change is to compare the error estimate of the decision tree after the pruning with that of the tree before the pruning:

$$D = E_{after} - E_{before} \tag{5.8}$$

From the available pruning alternatives, we choose the one where this difference is the smallest, D_{min}; but we carry out the pruning only if $D_{min} < c$, where c is a user-set threshold for how much performance degradation can be tolerated in exchange for the tree's compactness. The mechanism is repeated, with the decision tree becoming smaller and smaller, the stopping criterion being imposed by the constant c. Thus in Fig. 5.2, the first pruning step might have removed the subtree rooted at t_3; and the second might have removed the subtree rooted at t_6. At this

Table 5.8 Algorithm for decision tree post-pruning

```
c . . . a user-set constant

(1) Estimate the error rate of the original decision tree. Denote its value by $E_{before}$.
(2) Estimate the error rates of the trees obtained by alternative ways of pruning the original tree.
(3) Choose the pruning after which the estimated error rate experiences minimum increase,
    $D_{min} = E_{before} - E_{after}$, but only if $D_{min} < c$.
(4) Repeat steps (2) and (3) as long as $E_{before} - E_{after} < c$.
```

moment, the procedure was stopped because any further attempt at pruning resulted in a tree whose error estimate increased too much: the difference between the estimated error of the final pruned tree and that of the original tree on the left of Fig. 5.2 exceeded the user's threshold: $D > c$.

The principle is summarized by the pseudo-code in Table 5.8.

On-Line Pruning In the divide-and-conquer approach to tree-building, the subsequent tests divide the data into smaller and smaller subsets. Inevitably, the evidence supporting the choice of the tests at the lower tree-levels become weak—the reader will recall what Chap. 2 had to say about probabilities estimated based on insufficient data. In an extreme, if a tree node is reached by only two training examples, one positive and one negative, the two classes may by mere coincidence appear to be distinguished by some totally irrelevant attribute. Adding this test to the decision tree only increases the danger of training-set overfitting.

The motivation behind *on-line* pruning is to make sure this does not happen. Here is the rule: if the training subset is smaller than a user-specified minimum, m, stop further expansion of the tree.

Limits of Pruning In decision tree software, the extent of pruning is usually controlled by two parameters. For the needs of post-pruning, the constant c determines how much growth in the estimated error the user is willing to tolerate; for on-line pruning, the constant m determines the point where further training subset splitting should be prevented.

The main reason why pruning is recommended is that the removal of low-level tests (with poor statistical support) may reduce the danger of overfitting. This, however, will be the case only up to a certain point. In the extreme, strong pruning can result in a decision tree that has degenerated to a single leaf labeled with the majority class. Such classifier is of course useless.

Performance Consequences of Pruning Figure 5.3 illustrates the effect that pruning typically has on classification performance. The horizontal axis represents the extent of pruning as controlled by c or m or both. The vertical axis represents the error rate measured on the training set as well as the error rate measured on some future testing set (the latter consisting of examples that have *not* been used for learning, but whose class labels are known).

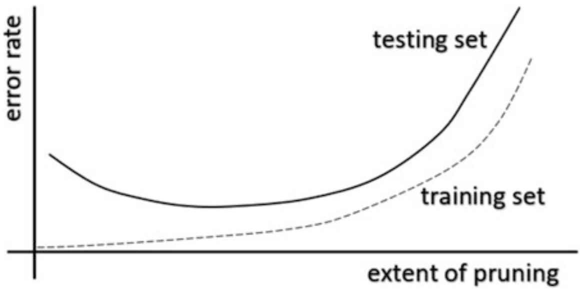


Fig. 5.3 With the growing extent of pruning (smaller trees), error rate on the testing set often drops, then starts growing again. Error rate on the training set usually increases monotonically

On the training set, error rate is minimized when there is no pruning at all—which is not surprising. More interesting is the testing-set curve whose shape indicates that an unpruned tree usually does poorly on testing data because of its tendency to overfit the training set, something that pruning is likely to reduce. Excessive pruning, however, removes attribute tests that *do* carry useful information, and the removal hurts future performance.

Some Observations Plotting these two curves in a concrete application domain may tell us a lot about the nature of the available data. In a noise-free domain and a relatively small training set, even very modest pruning will impair testing-set error rate. Conversely, a noisy domain is marked by a situation where pruning leads to improved classification on testing set.

Big distance between the two curves indicates that what has been learned from the training set is not going to be very useful in the future, perhaps because the training set did not contain the information necessary for the classification.

Finally, notice that the error rate on the testing set is almost always greater than the error rate on the training set.

5.5.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What are the potential benefits of decision tree pruning?
- How can we estimate the tree's error rate on future data? Write down the formula and explain how it is used.
- Explain the difference between post-pruning and on-line pruning.
- What parameters control the extent of pruning? How do they affect error rate on the training set, and error rate on the testing set?

5.6 Decision Tree Can Be Converted to Rules

One of the advantages of decision trees is that their behavior can be interpreted. Any sequence of tests along the path from the root to a leaf represents an *if-then* rule that explains why the classifier chose this or that concrete class label.

Rules Generated by a Decision Tree The reader will find it easy to convert a decision tree to a set of rules. It is enough to notice that a leaf is reached through a series of edges whose specific choice is determined by the results of the attribute tests encountered along the way. Each leaf is thus associated with a concrete conjunction of test results.

For illustration, let us write down the complete set of rules for the `pos` class as obtained from the decision tree in Fig. 5.1a.

if `crust-size=big AND filling-size=big` *then* `pos`

if `crust-size=big AND filling-size=small AND shape=circle` *then* `pos`

if `crust-size=small AND shape=circle` *then* `pos`

if `crust-size=small AND shape=(square OR triangle)`
`AND filling-size=big` *then* `pos`

else `neg`

Note the *default class*, `neg`, in the last line. An example is labeled with the default class if all rules fail, if the value of the *if*-part of each rule is *false*. In this two-class domain, it sufficed to write down the rules for the `pos` class, the other class being the default option. We could have done it the other way round, considering only the rules for the `neg` label, in which case `pos` will be the default class. The nature of the tree in Fig. 5.1a actually makes this other choice more economical: only two leaves are here labeled with `neg`, and this means only two rules. The reader may want to write down these two rules as a simple exercise.

The lesson is clear: in a domain with K classes, only the rules for $K - 1$ classes are needed, the last class being the default.

Pruning the Rules The tree post-pruning mechanism described earlier replaced a subtree with a leaf. This means that lower-level tests were the first to go, the technique being unable to remove a higher-level node before those below it. In on-line pruning, the situation is similar.

Once the tree has been converted to rules, however, pruning gains in flexibility: *any* test in the *if*-part of *any* rule is a potential candidate for removal; and entire rules can be deleted, too. This is done by the rule-pruning algorithm summarized by the pseudo-code in Table 5.9 and illustrated by the example in Table 5.10. Here, the initial set of rules was obtained from the tree in the left part of Fig. 5.2. The first

Table 5.9 Algorithm for rule-pruning

Re-write the decision tree as a set of rules.

Let c be a user-set constant controlling the extent of pruning

(1) In each rule, calculate how much the error estimate would increase after the removal of the individual tests.
(2) In each rule, choose the removal with the smallest error increase, D_{min}. If $D_{min} < c$, remove the test; otherwise, do not remove it.
(3) In the set of rules, search for the weakest rules to be removed, the term "weak" meaning that only a few training examples make the rule's left-hand side *true*.
(4) Choose the default class.
(5) Order the rules according to their strengths (how many training examples make the left-hand side *true*.).

Table 5.10 Illustration of the algorithm for rule-pruning

The decision on the left in Fig. 5.2 is converted into the following set of rules, **neg** being the default.

$t_1 \wedge t_2$	$\rightarrow$ pos
$t_1 \wedge \neg t_2 \wedge t_3 \wedge t_4$	$\rightarrow$ pos
$\neg t_1 \wedge t_5 \wedge t_6$	$\rightarrow$ pos
else neg	

Suppose the evaluation of the tests in the rules indicates that t_3 in the second rule and t_5 in the third rule can be removed without major increase in the error estimate. The two removals result in the following set of rules.

$t_1 \wedge t_2$	$\rightarrow$ pos
$t_1 \wedge \neg t_2 \wedge t_4$	$\rightarrow$ pos
$\neg t_1 \wedge t_6$	$\rightarrow$ pos
else neg	

The next step can reveal that the second (already modified) rule can be removed without a major increase in the error estimate because the rule is rarely employed. After its removal, the set of rules will look as follows.

$t_1 \wedge t_2$	$\rightarrow$ pos
$\neg t_1 \wedge t_6$	$\rightarrow$ pos
else neg	

This completes the pruning.

pruning step removes those tests that do not appear to contribute much to the overall classification performance; the next step deletes the rules that are rarely used.

We haste to admit that the price for this added flexibility is a significant increase in computational costs.

5.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the mechanism that converts a decision tree to a set of rules. How many rules are thus obtained? What is the motivation behind such conversion?
- What is a *default class*? Which class would you choose for default when converting a decision tree to a set of rules?
- Discuss the possibilities of rule-pruning. In what sense can we claim that rule-pruning offers more flexibility than the original decision tree pruning? What is the price for this increased flexibility?

5.7 Why Decision Trees?

Decision trees are nowadays less popular than they used to be, overshadowed as they are by other approaches that have gained popularity thanks to demonstrated classification performance. However, a conscientious engineer resists the dictates of fashion, focusing instead on the concrete benefits offered by this paradigm. Here are some ideas to consider.

Availability of Attribute Values First and foremost, the decision tree asks for one attribute value at a time. This is a great advantage in applications where the attribute values are not immediately available, and obtaining their values is difficult and/or expensive. Thus in medical diagnosis, laboratory tests are requested only as they are needed; to expect an exhaustive attribute vector containing all such tests right from the beginning would be absurd.

Domains of this kind virtually disqualify approaches such as k-NN or linear classifiers; they even render impractical the famous deep-leaning paradigm from Chap. 16. Under the circumstances, the engineer will gladly resort to good old decision trees.

Attribute Selection The spectrum of application possibilities of the mechanism that chooses the most informational attribute (see Sect. 5.3) goes well beyond the paradigm of decision trees. For instance, one of the sections in Chap. 11 discusses the need to identify attributes with the maximum contribution to the classification task at hand. The *filter*-based approach to address this task often relies on the information-theoretical formulas that are employed in the context of decision trees.

Creating Higher-Level Features Section 3.8 used the example of the concept of `kangaroo` to point out the limitations inherent in vectors of low-level attributes that lack the information necessary for classification. The conclusion was that machine learning perhaps needs some mechanisms to create higher-level features in the form of functions of the available attributes. As we will see in some of the

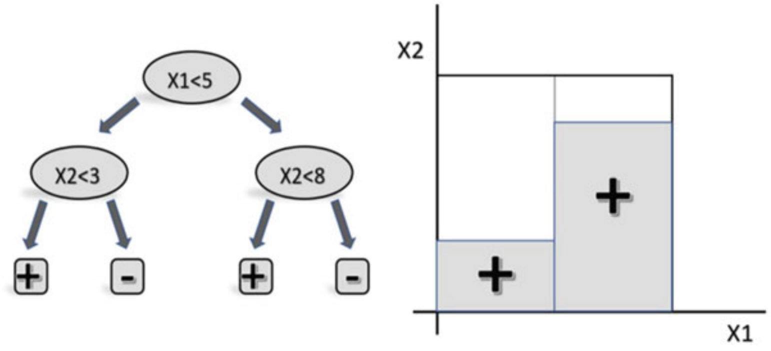


Fig. 5.4 The tests in the numeric decision tree on the left define the decision surface on the right. This decision surface consists of a set of axis-parallel line segments

upcoming chapters, the power of contemporary machine learning is often explained by implicit abilities to create those higher-level features.

One possibility is to interpret each branch of the induced decision tree as a new feature, defined by a conjunction of attribute-value tests.

Information-Based Loss in Neural Networks Section 5.3 introduced a formula to quantify the amount information contained in a message about an example's class. This formula is sometimes employed by algorithms to train artificial neural networks. For more about this, see Chap. 6.

Decision Tree Defines a Decision Surface The defining aspect of the linear and polynomial classifiers from Chap. 4 was their ability to induce a *decision surface* that separated examples from the positive class from those belonging to the negative class. A quick glance at Fig. 5.4 will convince us that decision tree, too, define decision surfaces of a different nature.

5.7.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the main benefit of decision trees in comparison to many other machine-learning paradigms?
- How can the knowledge of induction of decision trees help us in such tasks as attribute selection of higher-level feature creation?
- Explain the difference between the decision surface defined by a linear classifier and the decision surface defined by a decision tree.

5.8 Summary and Historical Remarks

- In decision trees, attributes are tested one at a time, the result of each test pointing to what should happen next: either another attribute test, or a class label if a leaf has been reached. One can say that a decision tree consists of a set of partially ordered tests, each sequence of tests defining one branch in the tree ending in a leaf.
- From the same training set, many alternative decision trees can usually be created. As a rule, smaller trees are to be preferred, their main advantages being interpretability, removal of irrelevant and redundant attributes, and reduced danger of overfitting noisy training data.
- The most common procedure for induction of decision trees from data proceeds in a recursive manner, always seeking to identify the attribute that conveys maximum information about the class label. Which attribute is "best" is determined by formulas borrowed from information theory. This approach tends to make the induced decision trees smaller.
- An important aspect of decision tree induction is pruning. The motivation is to make sure that all tests are supported by sufficient evidence. Pruning reduces the tree size which has obvious advantages (see above). Two types of pruning exist. (1) In post-pruning, the tree is first fully developed, and then pruned. (2) In on-line pruning (which may be a misnomer), the development of the tree is stopped once the training subsets used to determine the next test become too small. In both cases, the extent of pruning is controlled by user-set parameters (denoted c and m, respectively).
- A decision tree can be converted to a set of rules that can further be pruned, too. In a domain with K classes, it is enough to specify the rules for $K - 1$ classes, the remaining class being treated as the *default class*. The rules are usually easier to interpret than the original tree. Rule-pruning may lead to more compact classifiers, though at significantly increased computational costs.
- Perhaps the main advantage of decision trees is that, for classification, they do not require the availability of the entire attribute vector. This is beneficial in domains where the attribute values are not immediately available, and are difficult or expensive to obtain.
- Some of the techniques and formulas from this chapter will be found useful in the context of some tasks discussed in the upcoming chapters: neural-network training, attribute selection, and the creation of higher-level features.

Historical Remarks The idea underlying decision trees was first put forward by Hoveland and Hunt in the late 1950s. The work was later summarized in a book form by Hunt et al. (1966) who reported experience with several implementations of their Concept Learning System (CLS). Friedman et al. (1977) developed a similar approach independently. An early high point of the research was reached by Breiman et al. (1984) who developed the famous system CART. Their ideas were then imported to the machine-learning world by Quinlan (1979, 1986). For many years, the most popular implementation was C4.5 from Quinlan (1993). This chapter is based on a simplified version of C4.5.

5.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

5.9.1 Exercises

1. In Fig. 5.5, eight training examples are described by two attributes, `size` and `color`, the class being the material: `wood` or `plastic`.
   - What is the entropy of the training set when only the class labels are considered (ignoring attribute values)?
   - Using information gain (see Sect. 5.3), decide which of the two attributes is better at predicting the class
2. Take the decision tree from Fig. 5.1a and remove from it the bottom-right test on `filling-size`. Based on the training set from Table 5.1, what will be the error rate estimate before and after this "pruning"?
3. Choose one of the decision trees in Fig. 5.1 and convert it to a set of rules. Pick one of these rules and decide which of its tests can be removed with the minimum increase in the estimated error.
4. Consider a set of ten training examples with the following values of a certain continuous attribute: 3.6, 3.2, 1.2, 4.0, 0.8, 1.2, 2.8, 2.4, 2, 2, 1.0. Suppose that the first five of these examples, and also the last one, are positive, the remaining examples being all negative. What will be the best binary split of the domain of this attribute's values?

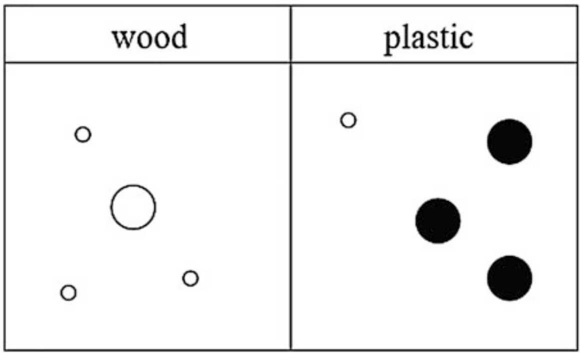


Fig. 5.5 A training set of circles described by two attributes: `size` and `color`. The class label is either `wood` or `plastic`

5.9.2 Give It Some Thought

1. The baseline performance criteria used for the evaluation of decision trees are error rate and tree size. (number of nodes). These, however, may not be appropriate in certain domains. Suggest applications where either the size of the decision tree or its error rate may be less important. Hint: consider the costs of erroneous decisions and the costs of obtaining attribute values.
2. What do you thing are the characteristics of a domain where a decision tree clearly outperforms the baseline 1-NN classifier? Hint: consider such characteristics as noise, irrelevant attributes, or the size of the training set; and then make your own judgement as to what impact each of them is likely to have on the classifier's behavior.
3. In what kind of data will a linear classifier outperform a decision tree? Suggest at least two features characterizing such data. Rely on the same hint as the previous question.
4. Having answered the previous two questions, you should be able to draw the logical conclusion: applying to the given data both decision trees and linear classification, what will their respective performances betray about the characteristics of the available data?
5. The decision tree approach described in this chapter gives only "crisp" yes-or-no classifications (one can argue that Bayesian classifiers are more flexible). Seeking to mitigate this limitation, how would you modify the decision trees framework so as to give, for each example, not only the class label, but also the classifier's confidence in this class label?

5.9.3 Computer Assignments

1. Implement the baseline algorithm for induction of decision trees and test its behavior on a few selected domains from the UCI repository.[4] Compare the results with those achieved by the k-NN classifier.
2. Implement the simple pruning mechanism described in this chapter. Choose a domain from the UCI repository. Run several experiments and observe how different extent of pruning affects the error rate on the training and testing sets.
3. Choose a sufficiently large domain from the UCI repository. Put aside 30% of the examples for testing. For training, use 10%, 20%, ... 70% of the remaining examples, respectively. Plot a graph where the horizontal axis gives the number of examples used for learning, and the vertical axis gives the computational time spent on the induction. Plot another graph where the vertical axis will give the error rate on the testing set. Discuss the results.

[4] www.ics.uci.edu/~mlearn/MLRepository.html.

4. Create your own synthetic data where examples are described by two numeric attributes with values from the unit interval, [0,1]. In the square $[0, 1] \times [0, 1]$, define a certain geometric figure (square, circle, ring, or something more complicated); all examples inside this figure are positive, those outside are negative. Induce a decision tree, then visualize its classification behavior using the following method. Create a testing set as a grid with all combinations of the two attributes (with increments 0.01); each of these points is then submitted to the decision tree. Display the results: the points labeled by the tree as positive are black and points labeled as negative are white. This is how the "positive" region is visualized. Show how the contours of these regions are for different domains affected by different amounts of pruning.

Chapter 6
Artificial Neural Networks

In *artificial neural networks*, many simple units, called *neurons*, are interconnected into larger structures. The network's classification behavior is determined by the weights of the links that interconnect the neurons. The task for machine learning is to provide algorithms capable of finding weights that result in good classification behavior. This search is accomplished by a process commonly referred to as a neural network's *training*.

Theory and practice of neural networks is too broad to be exhausted in by the twenty or thirty pages this book can afford to devote to them; it is necessary to narrow the focus. For this reason, only two types are discussed here as representatives of the whole field: multilayer perceptrons and radial-basis function networks. The latter are easy to implement and are known for high classification performance. The former have become highly topical thanks to their relation to the now-so-popular deep-learning paradigm.[1]

The chapter describes how to use these tools for classification and explains simple methods of their training. In the case of multilayer perceptrons, care is taken to lay grounds for the *deep learning* treated in a later chapter.

6.1 Multilayer Perceptrons

Let us simplify our job by assuming, for the time being, that all attributes are continuous and that they have been normalized into the interval $[-1, 1]$. Strictly speaking, normalization is not necessary, but it is certainly practical.

Neurons and *sigmoid* Function The function of a *neuron*, the basic unit of a multilayer perceptron, is simple: a weighted sum of the signals arriving at the input

[1]See Chap. 16.

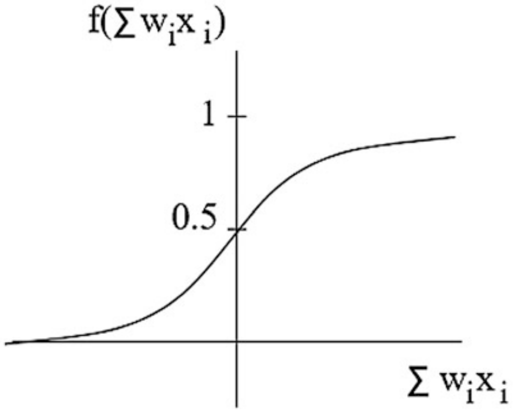


Fig. 6.1 Popular transfer function or early days of neural networks: `sigmoid`

is submitted to a *transfer function* (also known as *activation function*) that provides the neuron's output. Several different transfer functions can be used. To begin with, we will rely on the so-called *sigmoid* that is defined by the following formula, where Σ is the weighted sum of inputs:

$$f(\Sigma) = \frac{1}{1 + e^{-\Sigma}} \tag{6.1}$$

Figure 6.1 shows what sigmoid looks like. Note that $f(\Sigma)$ grows monotonically with the increasing value of Σ but never leaves the open interval (0, 1) because $f(-\infty) = 0$ and $f(\infty) = 1$. The vertical axis is intersected at f(0)=0.5. Common practice uses the same transfer function for all neurons in the network.

First Derivative of *sigmoid* Perhaps the main reason why *sigmoid* is in the field of neural networks so popular is that its first derivative can be expressed as a "function of the original function." It would be easy to prove the following:

$$f'(\Sigma) = f(\Sigma)[1 - f(\Sigma)] \tag{6.2}$$

What makes this formula so convenient will become clear once we proceed to the popular learning algorithm, Sect. 6.3.

First Derivative of *sigmoid* One of the reasons why *sigmoid* is so popular in neural networks is that it has a very convenient first derivative. Indeed, it would be easy to prove the following equality:

$$f'(\Sigma) = f(\Sigma)[1 - f(\Sigma)] \tag{6.3}$$

The convenience will become obvious once we proceed to the popular learning algorithm for neural networks (see Sect. 6.3).

Multilayer Perceptron (MLP) The neural network in Fig. 6.2 is known as *multilayer perceptron*. The neurons, represented by ovals, are arranged in two tiers:

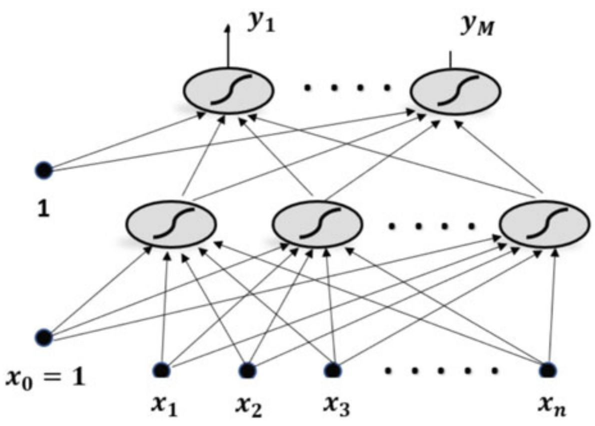


Fig. 6.2 A multilayer perceptron with two interconnected layers. The lower one is *hidden layer*, and the upper one is *output layer*

the *output layer* and the *hidden layer*.[2] For simplicity, we will now consider only networks with a single hidden layer while remembering that it is quite common to employ two such layers, even three, though rarely more than that. Note that the neurons (like the linear classifiers from Chap. 4) use also the zeroth weights to provide the neurons' trainable bias.

While there is no communication between neurons of the same layer, adjacent layers are fully interconnected. Importantly, each neuron-to-neuron link is associated with a *weight*. The weight of the link from the j-th hidden neuron to the i-th output neuron is denoted as $w_{ji}^{(1)}$ and the weight of the link from the k-th attribute to the j-th hidden neuron as $w_{kj}^{(2)}$. Note that the first index indicates where the link begins and the second tells us where it ends. The superscript, (1) or (2), indicates the layer.

Forward Propagation Suppose that the network's input has been presented with an example, $\mathbf{x} = (x_1, \dots, x_n)$. The attribute values are passed along the neural links that multiply each x_k by the weights associated with the corresponding links. For instance, the j-th hidden neuron receives as input the weighted sum, $\sum_k w_{kj}^{(2)} x_k$, and this sum is then subjected to the sigmoid activation function, $f(\sum_k w_{kj}^{(2)} x_k)$. The i-th output neuron then receives the weighted sum of the values arriving from the hidden neurons and, again, subjects the sum to the sigmoid activation function. This is how the network's i-th output is obtained. The process of propagating in this manner the attribute values from the network's input to its output is called *forward propagation*.

When using multilayer perceptrons for classification, we assign to each class one output neuron, and the value returned by the i-th output neuron is interpreted as evidence in support of the i-th class. For instance, if the values obtained at three

[2]When we view the network from above, the hidden layer is obscured by the output layer.

output neurons are $\mathbf{y} = (0.2, 0.6, 0.1)$, the classifier will label the given example with the second class because 0.6 is greater than 0.2 and 0.1.

In essence, the two-layer MLP represents the following Formula, where f is the sigmoid transfer function from Eq. 6.1, $w_{kj}^{(2)}$ and $w_{ji}^{(1)}$ are the weights of the links leading to the hidden and output layers, respectively, and x_k's denote the attribute values of the example presented to the network.

$$y_i = f(\sum_j w_{ji}^{(1)} f(\sum_k w_{kj}^{(2)} x_k)) \tag{6.4}$$

Numeric Example The principle is illustrated in Table 6.1. An example $\mathbf{x}$ is presented to the network's input. Before reaching hidden neurons, all attribute values are multiplied by the corresponding weights, and the weighted sums are subjected to activation functions. The results ($h_1 = 0.32$ and $h_2 = 0.54$) are then multiplied by the next layer of weights and forwarded to the output neurons where they are again subjected to the sigmoid function. This is how the two output values, $y_1 = 0.66$ and $y_2 = 0.45$, have been obtained. We see that the evidence supporting the class of the output neuron on the left is stronger than the evidence supporting the class of the output neuron on the right. The classifier therefore chooses the left neuron's class.

MLP Is a Universal Classifier Mathematicians have been able to prove that with the right choice of weights and with the right size of the hidden layer, Eq. 6.4 can approximate any realistic function with arbitrary accuracy. The consequence of this *universality theorem* is that the *multilayer perceptron* can in principle be used to address just about any classification task.

What the theorem *does not* tell us, however, is how many hidden neurons are needed and what the values of the individual weights should be. In other words, we know that the solution exists, but there is no guarantee we will ever find it.

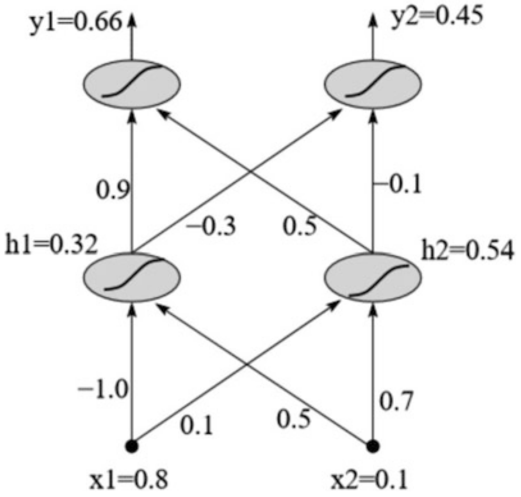
6.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain how an example described by a vector of continuous attributes is forward-propagated through the *multilayer perceptron*. How is the network's output interpreted when choosing the class label for the example?
- What is *transfer function* (also known as *activation function*)? Write down the formula defining the *sigmoid* transfer function and comment on its shape.
- What is the *universality theorem*? What does it inform us about, and what does it *fail* to tell us?

Table 6.1 Example of forward propagation in *multilayer perceptrons*

Task. Forward-propagate $\mathbf{x} = (x_1, x_2) = (0.8, 0.1)$ through the network below. To reduce the necessary calculations, the *bias* attribute, x_0, is ignored.



Solution.

Inputs of hidden-layer neurons:

$z_1^{(2)} = 0.8 \times (-1.0) + 0.1 \times 0.5 = -0.75$

$z_2^{(2)} = 0.8 \times 0.1 + 0.1 \times 0.7 = 0.15$

Outputs of hidden-layer neurons:

$h_1 = f(z_1^{(2)}) = \frac{1}{1+e^{-(-0.75)}} = 0.32$

$h_2 = f(z_2^{(2)}) = \frac{1}{1+e^{-0.15}} = 0.54$

Inputs of output-layer neurons:

$z_1^{(1)} = 0.32 \times 0.9 + 0.54 \times 0.5 = 0.56$

$z_2^{(1)} = 0.32 \times (-0.3) + 0.54 \times (-0.1) = -0.15$

Outputs of output-layer neurons:

$y_1 = f(z_1^{(1)}) = \frac{1}{1+e^{-0.56}} = 0.66$

$y_2 = f(z_2^{(1)}) = \frac{1}{1+e^{-(-0.15)}} = 0.45$

6.2 Neural Network's Error

Before introducing the technique for MLP's training, we need to take a closer look at how to quantify the accuracy of its classification decisions.

Error Rate and Its Limitation Let us begin by presenting to the *multilayer perceptron* an example, $\mathbf{x}$, whose known class is $c(\mathbf{x})$. Forward propagation results in labeling $\mathbf{x}$ with $h(\mathbf{x})$. If $h(\mathbf{x}) \neq c(\mathbf{x})$, the classification is incorrect. This may happen also to other examples, and the engineer wants to know *how often* this happens: he/she wants to know the *error rate*.

Error rate is calculated by dividing the number of errors by the number of examples that have been classified. For instance, if the classifier misclassifies 30 out of 200 examples, the error rate is $30/200 = 0.15$.

Error rate, however, paints but a crude picture of classification behavior. What it fails to reflect is the sigmoid's ability to measure the *size* of each error.

Numeric Example Suppose we are comparing two different neural networks, each with three output neurons that correspond to three classes, C_1, C_2, and C_3. Let us assume that, for a given example $\mathbf{x}$, one network outputs $\mathbf{y1}(\mathbf{x}) = (0.5, 0.2, 0.9)$ and the other $\mathbf{y2}(\mathbf{x}) = (0.6, 0.6, 0.7)$. Consequently, $\mathbf{x}$ will in either case be labeled with the third class, $h1(\mathbf{x}) = h2(\mathbf{x}) = C_3$.

If the correct answer is $c(\mathbf{x}) = C_2$, both networks got it wrong. However, the seriousness of the error is not the same. The reader will have noticed that the first network appears "very sure" about the class: 0.9 is clearly greater than the other two outputs, 0.5 and 0.2. By contrast, the second network seems rather undecided, the differences of the outputs (0.6, 0.6, and 0.7) being so small as to make the choice of C_3 appear almost arbitrary. In view of its weaker commitment to the incorrect class, the second network thus appears less mistaken than the first.

Situations of this kind are reflected in another performance criterion, the *mean squared error* (MSE). The next paragraphs will explain its principle.

Target Vector Before being able to define *mean squared error*, we have to introduce yet another important concept, the *target vector*. Let us denote by $\mathbf{t}(\mathbf{x})$ the target vector of example, $\mathbf{x}$. In a domain with m classes, the target vector, $\mathbf{t}(\mathbf{x}) = [t_1(\mathbf{x}), \ldots, t_m(\mathbf{x})]$, consists of m binary numbers. If the example belongs to the i-th class, then $t_i(\mathbf{x}) = 1$ and all other elements in this vector are $t_j(\mathbf{x}) = 0$ (where $j \neq i$).

Suppose that a given domain knows three classes, C_1, C_2, and C_3, and suppose that $\mathbf{x}$ belongs to C_2. The second neuron should output 1, and the other two neurons should output 0.[3] The target is therefore $\mathbf{t}(\mathbf{x}) = (t_1, t_2, t_3) = (0, 1, 0)$.

Mean Squared Error The idea is to quantify the differences between the output vector and the target vector:

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(t_i - y_i)^2 \tag{6.5}$$

To obtain the network's MSE, we need to establish for each output neuron the difference between its output and the corresponding element of the target vector. Note that the terms in the parentheses, $(t_i - y_i)$, are squared to make sure that negative differences are not subtracted from positive ones.

[3] More precisely, the outputs will only *approach* 1 and 0 because the sigmoid function is bounded by the *open* interval (0, 1).

Returning to the example of the two networks mentioned above, if the target vector is $\mathbf{t}(\mathbf{x}) = (0, 1, 0)$, then the mean squared errors have the following values:

$$MSE_1 = \tfrac{1}{3}[(0 - 0.5)^2 + (1 - 0.2)^2 + (0 - 0.9)^2)] = 0.57$$
$$MSE_2 = \tfrac{1}{3}[(0 - 0.6)^2 + (1 - 0.6)^2 + (0 - 0.7)^2] = 0.34$$

The reader can see that $MSE_2 < MSE_1$, which is in line with the intuition that the second network should be deemed "less wrong" on $\mathbf{x}$ than the first.

6.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- In what sense does error rate fail to provide adequate information about a neural network's classification performance?
- Explain the difference between a neural network's output, the example's class, and the target vector.
- Write down the formulas defining the *error rate* and the *mean squared error*.

6.3 Backpropagation of Error

In *multilayer perceptrons*, the network's behavior is determined by the sets of trainable weights, $w_{ji}^{(1)}$ and $w_{kj}^{(2)}$. The task for machine learning is to find weights that optimize classification performance. This is achieved by *training*.

General Scenario The procedure is analogous to the one employed by linear classifiers. At the beginning, the weights are *initialized* to small random numbers, typically from the interval $[-0.1, 0.1]$, perhaps even smaller. Training examples are then presented one by one, and each of them is forward-propagated to the output layer. The discrepancy between the network's output and the example's target vector determines how much to modify the weights (see below).

After weight modification, the next example is presented. When the last training example has been reached, one *epoch* has been completed. In *multilayer perceptrons*, the number of epochs needed for successful training is much greater than what it was in linear classifiers: it can be thousands, tens of thousands, even more.

Local Minima and Global Minima Figure 6.3 helps us grasp the problem's nature. The vertical axis represents the mean squared error, a function of the network's weights plotted along the horizontal axes. For convenience, we pretend that there are only two weights—which is unrealistic, but we cannot use more than two because we are unable to visualize multidimensional spaces.

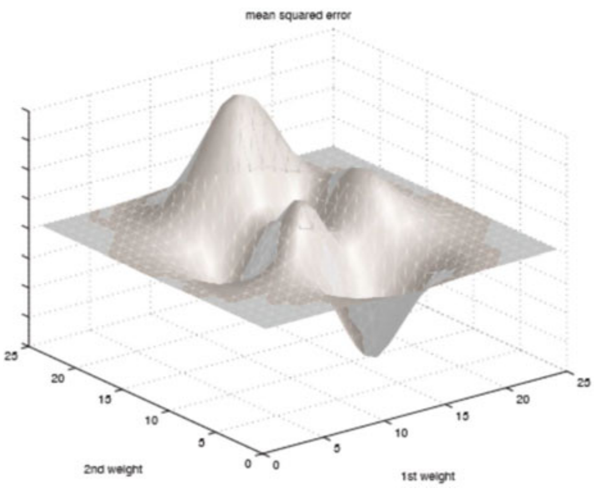


Fig. 6.3 For a given example, each set of weights results in a certain mean squared error. Training should reduce this error as quickly as possible, the steepest slope

The point is that the error function can be imagined as a kind of a "landscape" whose "valleys" represent the function's *local minima*. The deepest is the *global minimum*, and this is what the training procedure ideally should reach; more accurately, training wants to find the set of weights corresponding to the global minimum.

Gradient Descent A quick glance at Fig. 6.3 tells us that any pair of weights defines for the given training example a concrete location, typically somewhere on one of the slopes. Any weight change will result in different coordinates along the horizontal axes and thus a different value of the error function. Where exactly this new location is, whether "up" or "down" the slope, will depend on how much, and in what direction, each of the weights has changed. For instance, it may happen that increasing both w_1 and w_2 by 0.1 will lead only to a minor reduction of the mean squared error, whereas increasing w_1 by 0.3 and w_2 by 0.1 will reduce it considerably.

The technique described below seeks weight changes that lead to the *steepest descent* down the error function. In the language of mathematics, this is called *gradient descent*.

Backpropagation of Error The weight-adjusting formulas can be derived from Eq. 6.4 by finding the function's gradient. This book is meant for practitioners and not for mathematicians, so let us skip the derivation and focus instead on the procedure's behavior.

Intuitively, individual neurons differ in their contributions to the overall MSE: some of them spoil the game more than others. If this is the case, then the links leading to "bad" neurons should undergo more significant weight adaptations than the links leading to less offensive ones; neurons that do not do much harm do not need much change.

Each neuron's responsibility for the overall error is established quite easily. The concrete formulas depend on what transfer function has been used. If it is the `sigmoid` defined by Eq. 6.1, then the responsibility is calculated as follows:

$$\textit{Output-layer neurons} : \delta_i^{(1)} = y_i(1 - y_i)(t_i - y_i)$$

Note that $(t_i - y_i)$ is the difference between the network's i-th output and the corresponding target value. This difference is multiplied by $y_i(1 - y_i)$.

The latter term is actually the *sigmoid*'s first derivative (recall Eq. 6.2), and its minimum is reached when $y_1 = 0$ or $y_i = 1$—a "strong opinion" about whether $\mathbf{x}$ should or should not be labeled with the i-th class. The term is maximized when $y_i = 0.5$, in which case the "opinion" is deemed neutral. Note that the sign of $\delta_i^{(1)}$ depends only on $(t_i - y_i)$ because $y_i(1 - y_i)$ is always positive.

$$\textit{Hidden-layer neurons} : \ \delta_j^{(2)} = h_j(1 - h_j) \sum_i \delta_i^{(1)} w_{ji}^{(1)}$$

Responsibilities of hidden neurons are calculated by *backpropagating* the output neurons' responsibilities , $\delta_i^{(1)} w_{ji}$, obtained in the previous step. This is the role of the term $\sum_i \delta_i^{(1)} w_{ji}$. Note that each $\delta_i^{(1)}$ (the responsibility of the i-th output neuron) is multiplied by the weight of the link connecting the i-th output neuron and the j-th hidden neuron. The weighted sum is multiplied by, $h_j(1 - h_j)$, essentially the same term as the one used in the previous step, except that the place of y_i has been taken by h_j.

Weight Updates Now that we know the responsibilities of the individual neurons, we are ready to update the weights of the links that lead to them. Similarly as in the case of *perceptron learning*, an additive rule is used:

output-layer neurons: $w_{ji}^{(1)} := w_{ji}^{(1)} + \eta \delta_i^{(1)} h_j$
hidden-layer neurons: $w_{kj}^{(2)} := w_{kj}^{(2)} + \eta \delta_j^{(2)} x_k$

The size of weight correction is determined by $\eta \delta_i^{(1)} h_j$ or $\eta \delta_j^{(2)} x_k$. Two observations can be made. First, the neurons' responsibilities, $\delta_i^{(1)}$ and $\delta_j^{(2)}$, are multiplied by η, the *learning rate* which, theoretically speaking, is from the unit interval, $\eta \in (0, 1)$, but usually much smaller values are used, typically less than 0.1. Second, the results are multiplied by $h_j \in (0, 1)$ and $x_k \in [-1, 1]$, respectively. The correction is therefore small. Its real effect is relative. If the added term's value is, say, 0.02, then smaller weights, such as $w_{ij}^{(1)} = 0.01$, will be affected more significantly than greater weights such as $w_{ij}^{(1)} = 1.8$.

The whole training procedure is summarized by the pseudo-code in Table 6.2. The reader will benefit from taking a closer look at the numeric example in Table 6.3, which provides all the necessary details of how the weights are updated in response to a single training example.

Table 6.2 Backpropagation of error in an MLP with one hidden layer

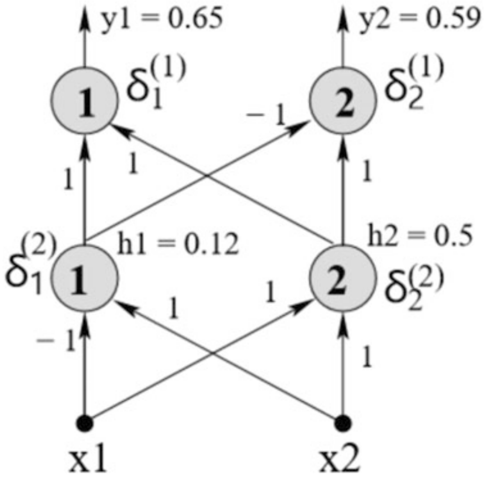
1. Present example **x** at the network's input and forward-propagate it.
2. Let $\mathbf{y} = (y_1, \dots y_m)$ be the output vector, and let $\mathbf{t}(\mathbf{x}) = (t_1, \dots t_m)$ be the target vector.
3. For each output neuron, calculate its responsibility, $\delta_i^{(1)}$, for the network's error:
   $\delta_i^{(1)} = y_i(1 - y_i)(t_i - y_i)$
4. For each hidden neuron, calculate its responsibility, $\delta_j^{(2)}$, for the network's error. While doing so, use the responsibilities, $\delta_i^{(1)}$, of the output neurons as obtained in the previous step.
   $\delta_j^{(2)} = h_j(1 - h_j) \sum_i \delta_i^{(1)} w_{ji}^{(1)}$
5. Update the weights using the following formulas, where η is learning rate:

   output layer: $w_{ji}^{(1)} := w_{ji}^{(1)} + \eta\delta_i^{(1)} h_j$; h_j: the output of the j-th hidden neuron
   hidden layer: $w_{kj}^{(2)} := w_{kj}^{(2)} + \eta\delta_j^{(2)} x_k$; x_k: the value of the k-th attribute

6. Unless a termination criterion has been satisfied, return to step 1.

Table 6.3 Illustration of backpropagation of error

Task. In the neural network below, let the transfer function be $f(\Sigma) = \frac{1}{1+e^{-\Sigma}}$. Using backpropagation of error (with $\eta = 0.1$), show how the weights are modified after the presentation of the following example: $[\mathbf{x}, \mathbf{t}(\mathbf{x})] = [(1, -1), (1, 0)]$. To reduce the necessary calculations, the *bias* attribute, x_0, is ignored.



Forward propagation.

The picture shows the state after forward propagation when the signals leaving the hidden and the output neurons have been calculated as follows:

- $h_1 = \frac{1}{1+e^{-(-2)}} = 0.12$
  $h_2 = \frac{1}{1+e^{0}} = 0.5$
  $y_1 = \frac{1}{1+e^{-(0.12+0.5)}} = 0.65$
  $y_2 = \frac{1}{1+e^{-(-0.12+0.5)}} = 0.59$

(the solution continues on the next page)

Backpropagation of error (cont. from the previous page)

The target vector being $\mathbf{t}(\mathbf{x}) = (1, 0)$ and the output vector $\mathbf{y} = (0.65, 0.59)$, the next step establishes each neuron's responsibility for the overall error. Here are the calculations for the output neurons:

- $\delta_1^{(1)} = y_1(1-y_1)(t_1-y_1) = 0.65(1-0.65)(1-0.65) = 0.0796$
  $\delta_2^{(1)} = y_2(1-y_2)(t_2-y_2) = 0.59(1-0.59)(0-0.59) = -0.1427$

Using these values, we calculate the responsibilities of the hidden neurons. Note that we will first calculate (and denote by Δ_1 and Δ_2) the backpropagated weighted sums, $\sum_i \delta_i^{(1)} w_{ij}^{(1)}$, for each of the two hidden neurons.

- $\Delta_1 = \delta_1^{(1)} w_{11}^{(1)} + \delta_2^{(1)} w_{12}^{(1)} = 0.0796 \times 1 + (-0.1427) \times (-1) = 0.2223$
  $\Delta_2 = \delta_1^{(1)} w_{21}^{(1)} + \delta_2^{(1)} w_{22}^{(1)} = 0.0796 \times 1 + (-0.1427) \times 1 = -0.0631$
  $\delta_1^{(2)} = h_1(1-h_1)\Delta_1 = 0.12(1-0.12) \times 0.2223 = 0.0235$
  $\delta_2^{(2)} = h_2(1-h_2)\Delta_2 = 0.5(1-0.5) \times (-0.0631) = -0.0158$

Once the responsibilities are known, weight modifications are straightforward:

- $w_{11}^{(1)} = w_{11}^{(1)} + \eta\delta_1^{(1)} h_1 = 1 + 0.1 \times 0.0796 \times 0.12 = 1.00096$
  $w_{21}^{(1)} = w_{21}^{(1)} + \eta\delta_1^{(1)} h_2 = 1 + 0.1 \times 0.0796 \times 0.5 = 1.00398$
  $w_{12}^{(1)} = w_{12}^{(1)} + \eta\delta_2^{(1)} h_1 = -1 + 0.1 \times (-0.1427) \times 0.12 = -1.0017$
  $w_{22}^{(1)} = w_{22}^{(1)} + \eta\delta_2^{(1)} h_2 = 1 + 0.1 \times (-0.1427) \times 0.5 = 0.9929$
- $w_{11}^{(2)} = w_{11}^{(2)} + \eta\delta_1^{(2)} x_1 = -1 + 0.1 \times 0.0235 \times 1 = -0.9977$
  $w_{21}^{(2)} = w_{21}^{(2)} + \eta\delta_1^{(2)} x_2 = 1 + 0.1 \times 0.0235 \times (-1) = 0.9977$
  $w_{12}^{(2)} = w_{12}^{(2)} + \eta\delta_2^{(2)} x_1 = 1 + 0.1 \times (-0.0158) \times 1 = 0.9984$
  $w_{22}^{(2)} = w_{22}^{(2)} + \eta\delta_2^{(2)} x_2 = 1 + 0.1 \times (-0.0158) \times (-1) = 1.0016$

After these weight updates, the network is ready for the next training example.

6.3.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Why is the training technique called "backpropagation of error"? Why do we need to establish the neurons' responsibilities?
- Discuss the behaviors of the formulas that calculate the responsibilities of the neurons in different layers.
- Explain the behaviors of the weight-updating formulas.

6.4 Practical Aspects of MLP's

To train *multilayer perceptrons* is more art than science. The engineer relies on experience and on deep understanding of MLP's various aspects. Let us briefly survey some of the more important features that need to be kept in mind.

Computational Costs Backpropagation of error is computationally expensive. Upon the presentation of an example, the responsibility of each individual neuron has to be calculated, and the weights modified accordingly. This is repeated for each training example, usually for many epochs. To get an idea of the real costs of all this, consider a network that is to classify examples described by 1000 attributes. If there are 100 hidden neurons, then the number of weights in this lower layer is $1000 \times 100 = 10^5$. This is how many weights have to be changed after each training example. Note that the upper-layer weights can be neglected as long as the number of classes is small. For instance, in a domain with three classes, the number of upper-layer weights is $100 \times 3 = 300$, which is much less than the 10^5 weights at the lower layer.

Suppose the training set consists of 10^4 examples, and suppose that the training will run for 10^4 epochs. The number of weight updates is then $10^5 \times 10^4 \times 10^4 = 10^{13}$. This is a whole lot, but nothing out of the ordinary; many applications are much more demanding. Methods of more efficient training have been developed, but their detailed study is beyond the scope of an introductory text.

***One-Hot* Representation of Discrete Attributes** Multilayer perceptrons expect continuous-valued attributes, a limitation that is easy to overcome. The values of Boolean attributes can be replaced with 1 for *true* and 0 for *false*. In the case of multi-valued discrete attributes, the so-called *one-hot* representation is quite popular. The idea is to replace the n-valued attribute with n binary attributes, of which one and only one is set to 1 ("hot"), and all the others are set to 0.

For illustration, `season` has four values: `spring`, `summer`, `fall`, and `winter`. *One-hot* representation replaces this attribute with four binary attributes, one for each season. In this 4-tuple, for instance, `spring` is represented as $(1, 0, 0, 0)$ and `fall` is represented as $(0, 0, 1, 0)$.

Batch Processing The weight adjustments do not have to be made after each training-example presentation. An alternative method sums the weight modifications as they are recommended at each example presentation but carries out the actual weight change only after the presentation of m training examples (where m is a user-set parameter) by adding to each trainable parameter the sum of the accumulated recommendations.

This *batch training* is not only computationally cheaper but also easier to implement because modern programming languages often support matrix operations.

Target Values Revisited For simplicity, we have so far assumed that each target value is either 1 or 0. This may not be the best choice. For one thing, these values can never be reached by a neuron's output, y_i because the range of the sigmoid function is the open interval (0, 1). Moreover, the weight changes in the vicinity of these two extremes are minuscule because the calculation of the output neuron's responsibility, $\delta_i^{(1)} = y_i(1 - y_i)(t_i - y_i)$, returns a value very close to zero whenever y_i approaches 0 or 1. Finally, we know that the classifier chooses the class whose output neuron has returned the highest value. The individual neuron's output precision therefore is not so important; what matters is the comparison with the other outputs. If the forward propagation results in $\mathbf{y} = (0.9, 0.1, 0.2)$, then the example is bound to be labeled with the first class (the one supported by $y_i = 0.9$), and this decision will not be affected by minor weight changes.

In view of these arguments, more appropriate values for the target are recommended: for instance, $t_i(\mathbf{x}) = 0.8$ if the example belongs to the i-th class, and $t_i(\mathbf{x}) = 0.2$ if it does not. Suppose there are three classes, C_1, C_2, and C_3, and suppose that $c(\mathbf{x}) = C_1$. In this case, the target vector will be $\mathbf{t}(\mathbf{x}) = (0.8, 0.2, 0.2)$. Both 0.8 and 0.2 find themselves in regions of relatively high sensitivity of the sigmoid function (in the curve's "knee") and as such will mitigate most of the concerns raised in the previous paragraph.

Local Minima Figure 6.3 illustrates the main drawback of the gradient-descent approach to MLP training. The weights are changed in a way that follows the steepest slope, but once the bottom of a *local minimum* has been reached, there is nowhere else to go—which is awkward: after all, the ultimate goal is to reach the *global minimum*. Two things are needed: first, a mechanism to tell a local minimum from a global one; second, a method to recover from having fallen into local minimum.

One way to identify local minima in the course of training is to keep track of the mean squared error (MSE) and to sum it up over the entire training set at the end of each epoch. Under normal circumstances, this sum tends get smaller as the training proceeds from one epoch to another. When it reaches a plateau where hardly any error reduction is observed, the learning process is suspected of being trapped in a local minimum.

Techniques to overcome this difficulty usually rely on adaptive learning rates (see the next paragraph) and on adding new hidden neurons (see Sect. 6.5). Generally speaking, the problem is less critical in networks with many hidden neurons. Also, local minima tend to be shallower, and less frequent, if all weights are very small, say, from the interval $(-0.01, 0.01)$.

Time-Dependent Learning Rate When describing backpropagation of error, we assumed a constant learning rate, η. In realistic applications, this is rarely the case. Quite often, the training starts with a higher value of η, which is then gradually decreased in time. Here is the motivation. At the beginning, the greater weight changes reduce the number of epochs, and they may even help the MLP to "jump over" some local minima. Later on, however, this large η might cause overshooting the global minimum, and this is why its value should be decreased.

If we express the learning rate as a function of time, $\eta(t)$, where t tells us how many training epochs have been completed, then the following negative-exponential formula will gradually reduce the learning rate (α is the slope of the negative exponential, and $\eta(0)$ is the learning rate's initial value):

$$\eta(t) = \eta(0)e^{-\alpha t} \tag{6.6}$$

Adaptive Learning Rate It should perhaps be noted that some advanced weight-changing formulas are capable of reflecting "current tendencies." For instance, one popular solution consists in trying to implement a "momentum": if the last two weight changes were in the same direction (both positive or both negative), it makes sense to increase the weight-changing step; conversely, if a positive change was followed by a negative change (of vice versa), the weight-changing step should be reduced so as to prevent overshooting.

Overtraining Sufficiently large *multilayer perceptrons* are capable of modeling *any* decision surface, and this makes them prone to overfitting the training set if the training ran too long. The reader will recall that overfitting typically means perfect classification of noisy training examples with disappointing performance on testing examples. The phenomenon is often referred to as the network's *overtraining*.

The problem is not so painful in the case of small MLPs that do not have so many trainable parameters. But as the number of hidden neurons increases, the network gains in flexibility, and overtraining/overfitting can be a reason for concern. However, as we will learn in the next section, this does *not* mean that we should always prefer small networks. Small networks have problems of their own.

Validation Set There is a simple method to detect overtraining. If the training set is big enough, we can afford to leave aside some 10–20% examples for a so-called *validation set*. The validation set is never used for backpropagation of error; it is here to help us observe how the performance on independent data is evolving.

After each epoch, the training is interrupted, and the current version of the MLP is tested on the validation set, At the beginning, the sum of mean squared errors on the validation data tends to go down, but only up to a certain moment; then, it starts growing again, alerting the engineer that the training now tends to overfit the data.

Another Activation Function: *t*anh For the sake of completeness, we have to mention that, instead of `sigmoid`, another activation function is sometimes used, in classical neural networks: the hyperbolic tangent, **tanh**. Let S be the `sigmoid` function. `tanh` is then defined as follows:

$$\texttt{tanh} = 2S - 1 \tag{6.7}$$

If the neural activation function is **tanh**, however, *different formulas for the backpropagation of error have to be used.* Not to complicate things, this book has focused on `sigmoid` and not on `tanh`.

6.4.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What can you say about the computational costs of the technique of backpropagation of error?
- Explain why this section recommended the values of the target vector to be chosen from {0.8, 0.2} instead of from {1,0}.

- Discuss the problem of local minima. Why do they pose a problem for training? How can we reduce the danger of getting trapped in one?
- What are the benefits of an adaptive learning rate? What formula has been recommended for it?
- What do you know about the danger that the training might result in overfitting the training data? What countermeasure would you recommend?

6.5 Big Networks or Small?

How many hidden neurons to use? If there are only one or two, the network will lack flexibility: it will be unable to develop a complicated decision surface, and its training will be hampered by local minima. At the other extreme, using thousands of hidden neurons means very high computational costs because of the need to train so many neurons. Besides, very large networks are excessively flexible and as such prone to overfit the training set. The two extremes being harmful, a compromise is needed.

Performance Versus Size Suppose you run the following experiment. The available set of pre-classified examples is divided into two parts, one for training, and the other for testing. Training is carried out by several neural networks, each with a different number of hidden neurons. The networks are trained until no reduction of the training-set error rate is observed. After this, the error rate on the testing data is measured.

Optimum Number of Neurons The results of the experiment from the previous paragraph will typically look something like those shown in Fig. 6.4. Here, the horizontal axis represents the number of hidden neurons, and the vertical axis represents the error rate measured on an independent testing set. Typically, the error

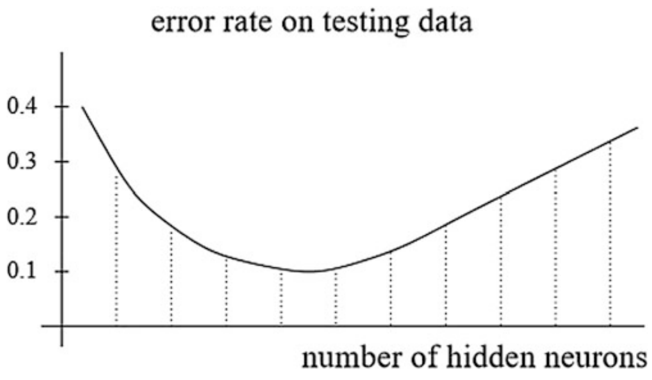


Fig. 6.4 Error rates on a testing set depend on the size of the hidden layer

rate will be high in the case of very small networks because small networks lack adequate flexibility and also suffer from the dangers posed by local minima. These weaknesses are mitigated if we increase the number of hidden neurons. As shown in the graph, these larger networks exhibit lower error rates. However, very large networks tend to overtrain and thus overfit the data. From a certain point, the testing-set error starts growing again (see the right tail of the graph).

The precise shape of this error curve depends on the complexity of the training data. In simple domains, the error rate is minimized when the network contains no more than 3–5 hidden neurons. In difficult domains, the minimum is reached only when hundreds of hidden neurons are employed. Also worth mentioning is the case where the training examples are completely noise-free. In this event, overfitting is less of an issue, and the curve's right tail may grow only moderately.

Search for Appropriate Size The scenario described above is for practical applications too expensive. After all, we have no idea whether we will need just a few neurons, or dozens of them, or hundreds, and we may have to rerun the computationally intensive training a great many times before establishing the size. It would thus be good to know some technique capable of finding the right size more efficiently.

One such technique is summarized by the pseudo-code in Table 6.4. The idea is to start with a very small network that only has a few hidden neurons. After each epoch, the learning algorithm checks the sum of the mean squared errors observed on the training set. The sum of errors is likely to decrease with the growing number of epochs—but only up to a certain point. When this is reached, the network's performance no longer improves, either because of its insufficient flexibility or because it got stuck in a local minimum. When this is observed, a few more neurons with randomly initialized weights are added, and the training is resumed.

Usually, the added neurons lead to further error reduction. In the illustration from Fig. 6.5, MSE levels off on two occasions; in both, adding new hidden neurons provided the necessary new flexibility.

Networks with More Than One Hidden Layer Up till now, only a single hidden layer has been considered. Practical experience shows, however, that, as far as computational costs and classification performance are concerned, better results

Table 6.4 Gradual search for appropriate size of the hidden layer

```
1. At the beginning, use only a few hidden neurons.
2. Train the network until the mean squared error no longer seems to improve.
3. At this moment, add a few neurons to the hidden layer, each with randomly initialized weights,
   and resume training.
4. Repeat the previous two steps until a termination criterion has been satisfied; for instance,
   when the new addition does not result in a significant error reduction, or when the hidden layer
   exceeds a user-set maximum size.
```

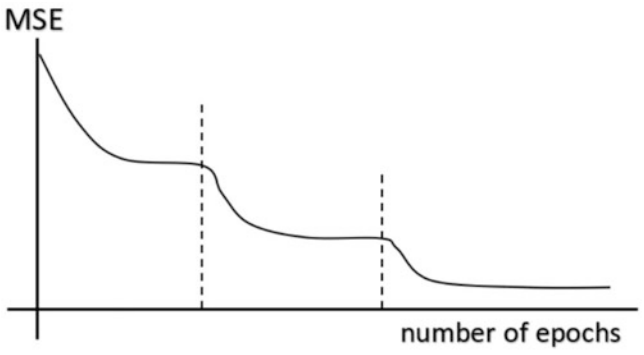


Fig. 6.5 When the training-set MSE does not seem to decrease, improvement may be possible by adding new hidden neurons. In this case, this happened twice

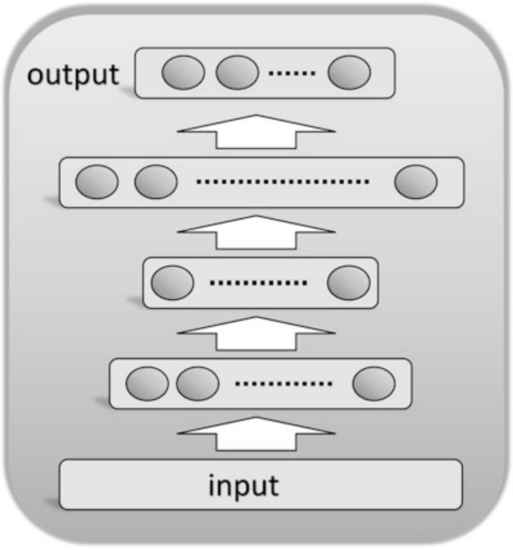


Fig. 6.6 Multilayer perceptron with three hidden layers

are sometimes achieved in neural networks with two or more hidden layers—the one in Fig. 6.6 has three hidden layers. The principles of forward and backward propagation remain virtually unaltered.

Let us introduce the formalism to be used in the training formulas. The reader will recall that previous sections referred to MLP's layers by superscripts. For instance, this convention denoted by $\delta_i^{(1)}$ the error responsibility of the i-th output neuron and by $w_{kj}^{(2)}$ the weight connecting the k-th attribute to the j-th hidden neuron.

In the case of networks with multiple hidden layers, we will refer to the output layer by superscript (1), the highest hidden layer by superscript (2), the next layer by superscript (3), and so on. Following this formalism, we will denote by $\delta_i^{(k)}$ the error responsibility of the i-th neuron in the k-th layer; the weights leading to this neuron will be denoted by $w_{ij}^{(k)}$, and the output of this i-th neuron will be denoted by $h_i^{(k)}$. Note that, with these denotations, the network's output is $y_i = h_i^{(1)}$.

Learning with Multiple Hidden Layers First of all, we use the output-layer errors to establish the error responsibilities of the output neurons in exactly the same way as in the case of the networks with a single hidden layer presented in Sect. 6.3.

The responsibilities thus obtained are then backpropagated all the way down to the input layer using the following generalization of the formula from Sect. 6.3:

$$\delta_j^{(k)} = \Sigma_i \delta_i^{(k-1)} w_{ji}^{(k-1)}$$

Once all neural responsibilities are known, they are used in the following weight modifications (as before, η is the learning rate, usually much smaller than in perceptrons):

$$w_{ji}^{(k)} = w_{ji}^{(k)} + \eta \delta_i^{(k)} h_j^{(k+1)}$$

Vanishing Gradient In the case of MLP's with sigmoid transfer function, experience has shown using more than two or three layers hardly ever adds to the networks' utility. The reason is known as the problem of *vanishing gradient*. Here is what it means.

In sigmoid transfer function, the output error values, $(t_i - y_i)$, can be very small, and the same happens at the lower layers. Moreover, the multiplier, $h_j^{(k)}(1 - h_j^{(k)})$, is small, as well. By consequence, the weight modification terms, $\Delta w_{ji}^{(k)}$, can become very small, almost imperceptible, as the value of the superscript k increases. When more than, say, three hidden layers are used, the learning process thus becomes computationally inefficient.

Concluding Comments on Multiple Hidden Layers The advantage of additional hidden layers is in the extra flexibility they offer. The disadvantage is the inefficiency of the learning process caused by the phenomenon of the vanishing gradient. Moreover, the additional layers usually increase the number of trainable parameters, and Chap. 7 will explain that the more trainable parameters we have, the more training examples are needed to reduce the danger of overtraining.

In the case of deep learning (see Chap. 16), many hidden layers are sometimes used. This is made possible not only by the much higher power of today's computers but also by the idea of using more appropriate transfer functions.

6.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Discuss the shape of the curve from Fig. 6.4. What are the shortcomings of very small and very large networks?
- Explain the algorithm that searches for a reasonable size of the hidden layer. What difficulties does it face?
- What are the advantages and disadvantages of using two or more hidden layers instead of just one? Explain the problem known as *vanishing gradient*.

6.6 Modern Approaches to MLP's

Over the last decade or so, the field of artificial neural networks has undergone dramatic evolution, culminating in the technology known as *deep learning*. Let us summarize some of the most important changes so as to prepare soil for Chap. 16.

Currently Popular Activation Functions One of the basic tenets of artificial neural networks is that the neural transfer functions have to be non-linear. During the first generations, hardly anybody doubted that the shape `sigmoid` is particularly useful, here. It was only recently that the scientific community realized that the non-linearities in much simpler piece-wise linear functions are just as good while offering some other advantages.

Figure 6.7 shows two currently popular activation functions. On the left is ReLU, which is an acronym for *Rectified Linear Unit*, and on the right is the LReLU, which is an acronym for *Leaky Rectified Linear Unit*. Both functions contain non-linearities, and both are differentiable, even if only piece-wise. Importantly, unlike `sigmoid`, none of them is upper-bounded.

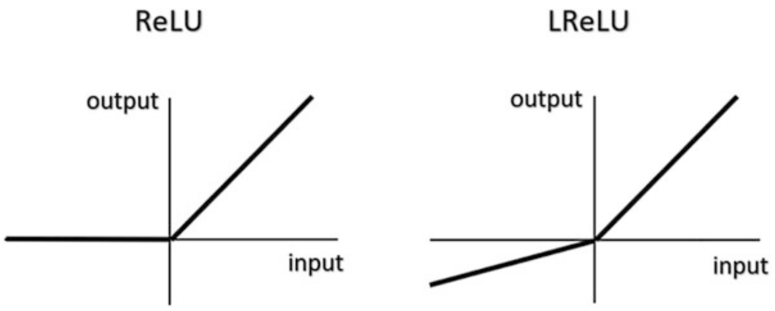


Fig. 6.7 Instead of sigmoids, modern neural networks prefer other activation functions, most typically ReLU (on the left) and LReLU (on the right)

Advantages of ReLU and LReLU Piece-wise linear functions are easy to differentiate. The first derivative at any interval of the input domain is a constant, sometimes even zero. The reader will agree that such derivatives are computationally cheaper and easier to backpropagate.

The fact that ReLU and LReLU do not suffer from the sigmoid's being imprisoned by the open interval (0, 1) not only allows the network to output values exceeding 1 but also virtually eliminates the problem of *vanishing gradient*. This means that networks with these activation functions can have many hidden layers without suffering from the main limitations of classical `sigmoid`-based MLP's.

Soft-Max: Turning Outputs to Probabilities Another recent innovation is certain post-processing of the network's output signals. This is motivated by the desire to interpret the outputs as probabilities that sum to 1.

Let $(y_1, \ldots, y_N)$ be the vector of the N outputs of the given neural network. The so-called *softmax* function recalculates the individual outputs by the following equation:

$$p_i = \frac{e^{y_i}}{\Sigma_j e^{y_j}} \tag{6.8}$$

It would be easy to verify that $\Sigma_i p_i = 1$.

Numeric Example Suppose that an MLP with three output neurons outputs the following vector: $y_1 = 0.3$, $y_2 = 0.8$, and $y_3 = 0.2$. Substituting these outputs in Eq. 6.8, we obtain the following values: $p_1 = 0.28$, $p_2 = 0.46$, and $p_3 = 0.26$. We can see that, indeed, $0.28 + 0.46 + 0.26 = 1.0$.

Information Loss Instead of MSE The fact that the outputs can now be interpreted as probabilities offers an interesting alternative to the way the network's error on a given example is measured. Instead of the mean squared error, a so-called *loss* function is often used. Here is the principle.

Suppose that an example belonging to class C_i has been presented at the input of an MLP and that this example's attribute values have been forward-propagated to the network's output. Suppose that after softmax, the i-the output (the one corresponding to class C_i) is p_i. Recalling what Chap. 5 had to say about how to measure information, the reader will agree that the amount of information conveyed by the message "the example belongs to the i-th class" can be calculated as follows:

$$L = -\log_2 p_i \tag{6.9}$$

In the language of machine learning, we say that the *loss* of the MLP upon the presentation of the given example is the L calculated by Eq. 6.9.

6.6.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Write the equations defining the ReLU and LReLU activation functions. What are their main advantages in comparison with the classical `sigmoid`?
- How is the *softmax* function defined? What are its advantages in comparison with the classical mean squared error?
- Define the neural network's *loss* function, and comment on its relation to information measurement.

6.7 Radial Basis Function Networks

Another popular class of neural networks is called *radial basis function* (RBF) network. Let us take a closer look.

Neurons Transform the Original Feature Space The behavior of an output neuron in a multilayer perceptron is similar to that of a linear classifier. This means that it may fare poorly in domains with classes that are not linearly separable. In the context of neural networks, however, this limitation is not necessarily harmful: the original examples have been transformed by the sigmoid functions in the hidden layer. Consequently, the neurons in the output layer deal with new "attributes," those obtained by this transformation. In the process of training, these transformed examples (the outputs of the hidden neurons) may become linearly separable so that the output-layer neurons can easily separate the classes.

Another Kind of Transformation There is another way of transforming the attribute values: by employing for the transfer function at hidden neurons the so-called radial-basis function (RBF). This is the case of the network depicted in Fig. 6.8. An example presented to the input is passed through a set of neurons that each return a value denoted here as φ_j.

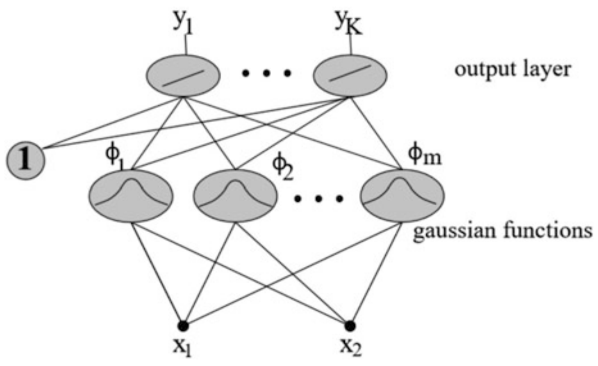


Fig. 6.8 Radial-basis function network

Radial-Basis Function (RBF) This is essentially the function used to model Gaussian distribution that we encountered in Chap. 2. Suppose that the attributes describing the examples all fall into some reasonably sized interval, say, $[-1, 1]$. For a given variance, σ^2, the following equation defines the n-dimensional Gaussian surface centered at $\boldsymbol{\mu}_j = [\mu_{j1}, \ldots \mu_{jn}]$ (the symbol "exp(x)" means "e to the power of x," where e is the base of natural logarithm):

$$\varphi_j(\mathbf{x}) = \exp\{-\frac{\Sigma_{i=1}^{n}(x_i - \mu_{ji})^2}{2\sigma^2}\} \tag{6.10}$$

RBF Networks In a sense, $\varphi_j(\mathbf{x})$ measures the similarity between the example vector, $\mathbf{x}$, and the Gaussian center, $\boldsymbol{\mu}_j$: the greater the distance between the two, the smaller the value of $\varphi_j(\mathbf{x})$. When vector $\mathbf{x}$ is to be classified, the network first transforms it to $\boldsymbol{\varphi}(\mathbf{x}) = [\varphi_1(\mathbf{x}), \ldots, \varphi_m(\mathbf{x})]$, where each $\varphi_j(\mathbf{x})$ measures the similarity of $\mathbf{x}$ to another center, $\boldsymbol{\mu}_j$.

The output signal of the i-th output neuron is $y_i = \sum_{j=0}^{m} w_{ji}\varphi_j(\mathbf{x})$, where w_{ji} is the weight of the link from the j-th hidden neuron to the i-th output neuron (the weights w_{0i} are connected to a fixed $\varphi_0 = 1$). This output signal being interpreted as the amount of evidence supporting the i-th class, the example is labeled with the i-th class if $y_i = \max_k(y_k)$.

One difference between the Bayesian classification and RBF networks is during the first reading often overlooked. The Naive Bayes classifier assumes all attributes to be mutually independent, and it can therefore work with the exponential functions of the scalar attributes. In RBF networks, however, the Gaussian centers are vectors, $\boldsymbol{\mu}_j = [\mu_{j1}, \ldots \mu_{jn}]$.

Output-Layer Weights It is easy to establish the output-layer *weights*, w_{ij}. Since there is only one layer of weights to be trained, we can use the *perceptron learning* algorithm from Chap. 4, applying it to examples whose descriptions have been transformed by the RBF functions of the hidden-layer neurons.

Gaussian Centers Common practice identifies the Gaussian centers, $\boldsymbol{\mu}_j$, with the individual training examples (again, note the relation to Bayesian classifiers). If the training set is small, we can simply use one hidden neuron per training example.

In many realistic applications, though, the training sets are large, and this can mean thousands of hidden neurons, even more. Realizing that impractically large networks can be unwieldy, many engineers prefer to select for the centers only a small subset of the training examples. Often, the choice is made at random. Another possibility is to find groups of similar vectors and then use for each RBF neuron the center of one group. Groups of similar vectors are discovered by the so-called *cluster analysis* that will be discussed in Chap. 15.

RBF-Based Support Vector Machines The RBF neurons transform the original example into a new vector of the transformed values, $\phi_1, \ldots \phi_m$. Most of the time, this transformation increases the chances that the newly described examples will be linearly separable. They can therefore be input to a linear classifier whose weights are trained by *perceptron learning*.

Let us mention here that it is quite popular to apply to the transformed examples the *support vector machine* introduced in Sect. 4.7. The maximized margins then optimize classification behavior on future examples. The resulting machine-learning tool is usually referred to as RBF-based SVM. Especially in domains where the inter-class boundaries are highly non-linear, this classifier is more powerful than the plain linear SVM.

Computational Costs RBF-network training consists of two steps. First, the centers of the radial-basis functions (Gaussian functions) are selected. Second, the output neurons' weights are obtained by perceptron training or by SVM. Since there is only one layer to train, the process is computationally less intensive than the training of *multilayer perceptrons*.

Relation to Nearest-Neighbor Classifiers Each Gaussian function represents a certain region in the instance space, and the output of this function quantifies a given example's distance from the center of this region. Suppose that each Gaussian is defined by one training example. In that event, the hidden layer transforms the original attributes into a feature vector that consists of the example's similarities to the individual training examples. The output neurons receive linear functions of these similarities.

In this sense, the RBF network can be seen as just another version of the nearest-neighbor approach. Many aspects of k-NN classifiers are equally relevant in RBF networks and as such have to kept in mind. Among these, perhaps the most important are irrelevant and redundant attributes, harmful and redundant examples, noise, and scaling (with the need to normalize attributes).

6.7.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of the radial-basis function network. In what aspects does it differ from the *multilayer perceptron*?
- How many weights need to be trained in an RBF network? What training approach would you use?
- What are the possibilities for the creation of the Gaussian functions in the hidden layer?

6.8 Summary and Historical Remarks

- The basic unit of *multilayer perceptrons (MLP)* is a *neuron*. The neuron receives a weighted sum of inputs and subjects this sum to a *transfer function* (also known as *activation function*). Several alternative transfer functions have been used.

Classical is the `sigmoid` defined by the following equation, where Σ is the weighted sum of inputs:

$$f(\Sigma) = \frac{1}{1 + e^{-\Sigma}}$$

Usually, all neurons use the same transfer function.

- The simplest version of an MLP consists of one output layer and one hidden layer of neurons. Neurons in adjacent layers are fully interconnected. There are no connections between neurons in the same layer. An example presented at the network's input is *forward-propagated* to its output, implementing, in principle, the following function:

$$y_i = f(\sum_j w_{ji}^{(1)} f(\sum_k w_{kj}^{(2)} x_k))$$

  Here, $w_{ji}^{(1)}$ and $w_{kj}^{(2)}$ are the weights of the output neurons and the hidden neurons, respectively, and f is the activation function.
- MLP training is accomplished by *backpropagation of error*. For each training example, the technique first establishes each neuron's responsibility for the network's overall error and then updates the weights according to these responsibilities.

  Here is how the responsibilities are calculated:

  output neurons: $\delta_i^{(1)} = y_i(1 - y_i)(t_i - y_i)$
  hidden neurons: $\delta_j^{(2)} = h_j(1 - h_j) \sum_i \delta_i^{(1)} w_{ij}$

  Here is how the weights are updated:

  output layer: $w_{ji}^{(1)} := w_{ji}^{(1)} + \eta \delta_i^{(1)} h_j$
  hidden layer: $w_{kj}^{(2)} := w_{kj}^{(2)} + \eta \delta_j^{(2)} x_k$

- Among the practical aspects of MLP training that the engineer has to consider, the most serious are excessive computational costs, existence of local minima, the need for adaptive learning rate, the danger of overfitting, and the size of the hidden layer.
- When more than one hidden layer is used, essentially the same learning formulas are used. The MLP is then more flexible but suffers from its own shortcomings such as excessive computational costs and *vanishing gradient* (the lower the hidden layer, the weaker the learning signal).
- Modern approaches to MLP differ from classical ones in three main aspects. First, activation functions ReLU and LReLU are used instead of `sigmoid`. Second, the output is subjected to the *softmax* function that makes it possible to interpret the network's outputs as probabilities. Third, the network's error is measured by an information-based *loss* function instead of the mean squared error.

- An alternative to MLP is the *radial-basis function* (RBF) network. For the transfer function at the hidden-layer neurons, the Gaussian function is used. The output-layer neurons often use the step function (in principle, a linear classifier) or simply a linear function of the inputs.
- In RBF networks, each Gaussian center is identified with one training example. If there are too many training examples, a random choice is sometimes made. Alternatively, cluster analysis can be used, and each neuron is then associated with one of the discovered clusters. If the domains of the original attributes have been normalized to $[-1, 1]$, the Gaussian variances are usually set to $\sigma^2 = 1$.
- The output-layer neurons in RBF networks can be trained by perceptron learning. Only one layer of weights needs to be trained, and this makes RBF networks computationally more affordable than MLPs.
- Sometimes, *support vector machines* (SVMs) are applied to the outputs of the hidden neurons. The resulting tool is known as RBF-based SVM, and it tends to give better results than when perceptron training is used—though at the price of higher computational costs.
- The hidden layers in MLP's and the Gaussian functions in RBF networks can be seen as useful methods of creating meaningful higher-level features from the original attributes. In the case of RBF networks, the new features are linear functions of the example's similarities to the individual training examples.

Historical Remarks Research of neural networks was famously delayed by the skeptical views expressed by Minsky and Papert (1969). The pessimism voiced by these famous scholars was probably the main reason why an early version of neural-network training by Bryson and Ho (1969) was largely overlooked, a fate soon to be shared by an independent successful effort by Werbos (1974). It was only after the publication of the groundbreaking volumes by Rumelhart and McClelland (1986), where the algorithm was independently reinvented, that the field of artificial neural networks became a respectable scientific discipline. More specifically, they popularized the rediscovery by Rumelhart et al. (1986).[4] The gradual growth of the multilayer perceptron was proposed by Ash (1989). The idea of radial-basis functions was first cast in the neural-network setting by Broomhead and Lowe (1988).

6.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

[4]The authors honestly acknowledge that this was a rediscovery.

6.9.1 Exercises

1. Return to the illustration of backpropagation of error in Table 6.3. Using only a pen, a paper, and a calculator, repeat the calculations for a slightly different training example: $\mathbf{x} = (-1, -1)$, $t(\mathbf{x}) = (0, 1)$.
2. Hand-simulating backpropagation of error as in the previous example, repeat the calculation for the following weight initializations:
   high output-layer weights: $w_{11}^{(1)} = 3.0$, $w_{12}^{(1)} = -3.0$, $w_{21}^{(1)} = 3.0$, $w_{22}^{(1)} = 3.0$
   small output-layer weights: $w_{11}^{(1)} = 0.3$, $w_{12}^{(1)} = -0.3$, $w_{21}^{(1)} = 0.3$, $w_{22}^{(1)} = 0.3$
   Observe the relative changes in the weights in each case.
3. Consider a training set containing of 10^5 examples described by 1000 attributes. What will be the computational costs of training a *multilayer perceptron* with 1000 hidden neurons and 10 output neurons for 10^5 epochs?

6.9.2 Give It Some Thought

1. Suggest a generalization of backpropagation of error for a *multilayer perceptron* with more than one hidden layer.
2. Section 6.1 suggested that all attributes should be normalized here to the interval $[-1.0, 1.0]$. How will the network's classification and training be affected if the attributes are not normalized in this way? (Hint: this has something to do with the sigmoid function.)
3. Discuss the similarities and differences of the classification procedures used in radial-basis functions and those used in *multilayer perceptrons*.
4. Compare the advantages and disadvantages of radial-basis function networks when compared to *multilayer perceptrons*.

6.9.3 Computer Assignments

1. Write a program that implements backpropagation of error for a predefined number of output neurons and hidden neurons. Use a fixed learning rate, η. Apply this program to selected benchmark domains from the UCI repository.[5] Experiment with different values of η, and see how they affect the speed of convergence.
2. For a given data set, experiment with different numbers of hidden neurons in the *multilayer perceptron*, and observe how they affect the network's ability to learn.

[5] www.ics.uci.edu/~mlearn/MLRepository.html.

3. Again, experiment with different numbers of hidden neurons. This time, focus on computational costs. How many epochs will the network need to converge? Observe also how error rate changes with time.
4. Write a program that for a given training set creates a radial-basis function network. For large training sets, select at random the examples to define the Gaussian centers. Apply the program to selected benchmark domains from the UCI repository.
5. Create an artificial two-dimensional domain with two attributes, both from the unit interval, [0, 1]. In this unit square, define a certain region such as a circle, rectangle, or perhaps something more complex. Label all examples inside this region as positive and all examples outside this region as negative. Then train an MLP on these training data. Then create an "exhaustivw" testing set where for each combination of the two values (minimum step 0.01) the points labeled by the induced network as positive are black and points labeled as negative are white. This will help visualize the shape of the region of positive examples. Show this region evolves in time (say, after each 100 training epochs). Experiment with different sizes of the hidden layer.

Chapter 7
Computational Learning Theory

As they say, nothing is more practical than a good theory. And indeed, mathematical models of learnability have helped improve our understanding of what it takes to induce a useful classifier from data, and, conversely, why the outcome of a machine-learning undertaking often disappoints the user. And so, even though this textbook does not want to be mathematical, it cannot help discussing at least the basic concepts of the *computational learning theory*.

At the core of this theory is the idea of *PAC-learning*, a paradigm that makes it possible to quantify learnability. Restricting itself to domains with noise-free discrete attributes, the first section of this chapter derives a simple expression that captures the mutual relation between the training-set size and the induced classifier's error rate. Practical consequences of this formula are briefly discussed in the two sections that follow. For domains with continuous attributes, the idea of the VC-dimension is then introduced.

7.1 PAC Learning

Perhaps the most useful idea behind the computational learning theory is the one of "probably approximate learning," sometimes abbreviated as *PAC learning*. Let us explain the underlying principles and then derive a formula that will provide some useful guidance.

Assumptions and Definitions The analysis will be easier if we build it around a few simplifying assumptions. First, the training examples—and also all future examples—are completely noise-free. Second, all attributes are discrete (none of them is continuous-valued). Third, the classifier acquires the form of a logical expression of attribute values; the expression is *true* for positive examples and *false* for negative examples. Finally, there exists at least one expression that correctly classifies all training examples.[1]

Each of the logical expressions can then be regarded as one hypothesis about what the classifier should look like. Together, the hypotheses form a *hypothesis space* whose size (the number of distinct hypotheses) is $|H|$. Under the assumptions listed above, $|H|$ is a finite number.

Inaccurate Classifier May Still Succeed on the Training Set Available training data rarely exhaust all subtleties of the underlying class. A classifier that labels correctly all training examples may still perform poorly in the future. The frequency of these mistakes may drop if we add more training examples because these additions may reflect aspects that were not represented in the original data. Here is the rule of thumb: the more training examples we have, the better the classifier induced from them.

How many training examples will give us a decent chance of future success? To find the answer, we will first consider a hypothetical classifier whose error rate on the entire instance space is greater than some predefined ϵ. Put another way, the probability that this classifier will label correctly a randomly picked example is less than $1 - \epsilon$. Taking this reasoning one step further, the probability, P, that this imperfect classifier will label correctly m random examples is bounded by the following expression:

$$P \leq (1 - \epsilon)^m \tag{7.1}$$

Here is what it means: with probability $P \leq (1 - \epsilon)^m$, an entire training set consisting of m examples will be correctly classified by a classifier whose error rate actually exceeds ϵ. Of course, this probability is for a realistic m very low. For instance, if $\epsilon = 0.1$ and $m = 20$ (which is a small set indeed), we will have $P < 0.12$. If we increase the training-set size to $m = 100$ (while keeping the error rate bounded by $\epsilon = 0.1$), then P drops to less than 10^{-4}. True, this is a very small number; but low probability is not impossibility.

Eliminating Poor Classifiers Suppose that an error rate greater than ϵ is deemed unacceptable. What are the chances that a classifier with performance as poor as that will be induced from the given training set (in other words, that it classifies correctly all training examples)?

The hypothesis space consists of $|H|$ classifiers. Let us consider the theoretical possibility that we evaluate all these classifiers on the m training examples and then retain only those classifiers that have never made a mistake. Among these "survivors," some will disappoint in the sense that, while being error-free on the training set, their error rates on the entire instance space actually exceed our tolerance, ϵ. Let there be k such offending classifiers.

[1]The attentive reader has noticed that all these requirements were satisfied by the "pies" domain from Chap. 1.

The concrete value of k cannot be established without evaluating each single classifier on the entire instance space. This being in the real world impossible, all we can say is that $k \leq |H|$, which is better because $|H|$ is known in many realistic cases, or at least can be calculated.[2]

Let us rewrite the upper bound on the probability that at least one of the k offending classifiers will be error-free on the m training examples.

$$P \leq k(1-\epsilon)^m \leq |H|(1-\epsilon)^m \tag{7.2}$$

This last expression establishes an upper bound on the probability that m training examples will succeed in eliminating all classifiers whose error rate on the entire instance space exceeds ϵ.

To become useful, the last expression has to be modified. We know from mathematics that $1-\epsilon < e^{-\epsilon}$, which means that $(1-\epsilon)^m < e^{-m\epsilon}$. With this in mind, we will express the upper bound in exponential form:

$$P \leq |H| \cdot e^{-m\epsilon} \tag{7.3}$$

Suppose we want this probability to be lower than some user-set δ:

$$|H| \cdot e^{-m\epsilon} \leq \delta \tag{7.4}$$

Taking the logarithm of both sides, and rearranging the terms, we obtain the formula that we will work with in the next few pages:

$$m > \frac{1}{\epsilon}(\ln|H| + \ln\frac{1}{\delta}) \tag{7.5}$$

Probably Approximately Correct (PAC) **Learning** The reason why we took the trouble of going through this derivation is that the reader thus gets a better grasp of the meanings and interpretations of the variables that may otherwise be a bit confusing. For quick reference, these variables are summarized in Table 7.1.

We are now able to define some important concepts. A classifier with error rate under ϵ is deemed *approximately correct*, and δ is the *probability* that this

Table 7.1 Variables involved in the analysis of PAC-learnability

m	...	The number of training examples
$\|H\|$	...	The size of the hypothesis space
ϵ	...	The classifier's maximum permitted error rate
δ	...	The probability that a classifier with error rate greater than ϵ is error-free on the training set

[2]Recall that in the "pies" domain from Chap. 1, the size of the hypothesis space was $|H| = 2^{108}$. Of these hypotheses, 2^{96} classified correctly the entire training set.

approximately correct classifier will be induced from m training examples (m being a finite number). Hence the name of the whole paradigm: probably approximately correct learning, or simply *PAC learning*. For the needs of this chapter, we will say that a concept is *not PAC-learnable* if the number of examples needed to satisfy the given (ϵ, δ)-requirements is so high that we cannot expect a training set of this size ever to be available—or, if it *is* available, that the learning software will need impractically long time (say, thousands of years) to induce from it the classifier.

Interpretation Inequality 7.5 specifies how many training examples, m, are needed if, with probability at least δ, a classifier with error rate below ϵ is to be induced. Note that this result does not depend on the concrete machine-learning technique. It depends only on the size, $|H|$, of the hypothesis space defined by the given type of classifier.

An important circumstance to remember is that m grows linearly in $1/\epsilon$. For instance, if we strengthen the limit on the error rate from $\epsilon = 0.2$ to $\epsilon = 0.1$, we will need (at least in theory) twice as many training examples to have the same chance, δ, of success. At the same time, the reader will also notice that m is less sensitive to changes in δ, growing only with the logarithm of $1/\delta$.

This said, we must not forget that our derivation was something like a worst-case analysis. As a result, the bound it has given us is less tight than a perfectionist might desire. For instance, the derivation allowed the possibility that $k = |H|$, which is clearly too pessimistic. Inequality 7.5 should thus never be interpreted as telling us how many training examples to use. It only provides a guidance that allows us to compare the learnability of alternative classifier types. We will pursue this idea in the next section.

7.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What are the meanings of the four variables (m, $|H|$, ϵ, and δ) used in Inequality 7.5?
- What does the term *PAC learning* refer to? Under what circumstances do we say that a class is *not PAC-learnable*?
- Derive Inequality 7.5. Discuss the meaning and practical consequences, and explain why it should only be regarded as a worst-case analysis.

7.2 Examples of PAC-Learnability

Inequality 7.5 tells us how learnability, defined by the (ϵ, δ)-requirements, depends on the size of the hypothesis space. Let us illustrate its use on two concrete classifiers.

Conjunctions of Boolean Attributes: The Size of the Hypothesis Space Suppose that all attributes are Boolean, that all data are noise-free, and that an example's class is known to be determined by a logical conjunction of attribute values: if *true*, then the example is positive, otherwise it is negative. For instance, the labels of the training examples may be determined by the following expression:

```
att1 = true AND att3 = false
```

This expression will label an example as positive if the value of the first attribute is *true* and the value of the third attribute is *false*, regardless of any other attribute (the other attributes are irrelevant for the classification). An example that fails to satisfy these two conditions will be labeled as negative.

Note that only conjunctions (ANDs) are permitted here, and that this constraint means that there are no ORs and no parentheses. The task for machine learning is to find an expression that satisfies this constraint and correctly labels all training examples. The set of all conjunctions permitted by the given language forms the hypothesis space, $|H|$. What is the size of this space?

In a logical conjunction of the kind specified above, each attribute is either *true* or *false* or irrelevant. This gives us three possibilities for the first attribute, times three possibilities for the second, and so on, times three possibilities for the last, n-th, attribute. The size of the hypothesis space is therefore $|H| = 3^n$.

Conjunctions of Boolean Attributes: PAC-Learnability Suppose that a noise-free training set is presented to machine-learning software capable of inducing classifiers of the just-defined form. To satisfy the last of the assumptions from the beginning of Sect. 7.1, we assume that at least one logical conjunction classifies correctly all training examples.

Since $\ln |H| = \ln 3^n = n \ln 3$, we rewrite Inequality 7.5 as follows:

$$m > \frac{1}{\epsilon}(n \ln 3 + \ln \frac{1}{\delta}) \tag{7.6}$$

We have obtained a conservative lower bound on the number of training examples that are needed if our (ϵ, δ)-requirements are to be satisfied: with probability δ, the induced classifier (error-free on the training set) will exhibit error rate less than ϵ on the entire instance space.

Note that the value of this expression grows linearly in the number of attributes, n. Theoretically speaking, then, if n is doubled, then twice as many training examples will be needed if, with probability limited by δ, classifiers with error rates above the predefined ϵ are to be weeded out.

Any Boolean Function: The Size of the Hypothesis Space Let us now investigate a broader class of classifiers, namely those defined by *any* Boolean function, allowing for all three basic logical operators (AND, OR, and NOT) as well as any number of parentheses. As before, we will assume that the examples are described by n Boolean attributes, and that they are noise-free.

What is the size of this hypothesis space?

From n Boolean attributes, 2^n different examples can be created. This defines the size of the instance space. For any subset of these 2^n examples, there exists at least one logical function that is *true* for all examples from this subset (labeling them as positive) and *false* for all examples from outside this subset (labeling them as negative). Two logical functions are regarded as identical from the classification point of view if both of them label any example with the same class; that is, if they never differ in their "opinion" about any example's class. The number of logical functions mutually distinct in their classification behavior is the same as the number of the subsets of the instance space.

A set consisting of X elements is known to have 2^X subsets. Since our specific instance space consists of 2^n examples, the number of its subsets is 2^{2^n}—and this is the size of our hypothesis space:

$$|H| = 2^{2^n} \tag{7.7}$$

Any Boolean Function: PAC-Learnability Since $\ln|H| = \ln 2^{2^n} = 2^n \ln 2$, Inequality 7.5 acquires the following form:

$$m > \frac{1}{\epsilon}(2^n \ln 2 + \ln\frac{1}{\delta}) \tag{7.8}$$

We conclude that the lower bound on the number of the training examples that are needed if the (ϵ, δ)-requirements are to be satisfied grows here exponentially in the number of attributes, n.

Such growth is prohibitive for any realistic value of n. For instance, even if we add only a single attribute, so that we now have $n + 1$ attributes, the value of $\ln|H|$ will double because $\ln|H| = 2^{n+1}\ln 2$ which is twice as much as $2^n \ln 2$. And if we add ten attributes, $n + 10$, then the value of $\ln|H|$ increases a thousand times because $\ln|H| = 2^{n+10}\ln 2 = 2^{10} \cdot 2^n \ln 2 = 1,024 \cdot 2^n \cdot \ln 2$. In other words, we would need a thousand times larger training set.

This observation is enough to convince us that a classifier in this general form is *not PAC-learnable*.

Word of Caution We must be careful not to jump to conclusions. As pointed out earlier, the derivation of Inequality 7.5—a worst-case analysis of sorts—relied on simplifying assumptions that render the obtained bounds rather conservative. In reality, the number of the training examples needed for the induction of a reliable classifier is much smaller than what our "magic formula" suggested. This said, the formula does offer evidence that an attempt to induce the "any Boolean function" will be extremely sensitive to the presence of irrelevant attributes, whereas the "conjunctions-only" classifier is less demanding.

For the engineer seeking to choose an appropriate learning technique, Inequality 7.5 thus offers a way of comparing PAC-learnability of classifiers constrained by different "languages." In our case, we saw that a conjunction of attribute values can be learned from a reasonably-sized training set, whereas a general concept defined by *any* Boolean function usually cannot.

Some other corollaries will be discussed in Sect. 7.3.

7.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the size of a hypothesis space consisting of conjunctions of attribute values? Substitute this size to Inequality 7.5 and discuss the result.
- Can you find the value of $|H|$ for some other class of classifiers?
- Explain, in plain English, why a Boolean function in its general form is not *PAC-learnable*.

7.3 Practical and Theoretical Consequences

The analysis in this chapter's first section offers a broader perspective of the learning task; a closer look at its results discovers practical clues whose benefits range from purely intellectual satisfaction of the theoretician to practical guidelines appreciated by the down-to-earth engineer. Let us take a quick look.

Bias and Learnability Section 7.2 explored *PAC-learnability* of classes whose descriptions are limited to conjunctions of attribute values. Such constraint represents a *bias* toward a specific type of classifier.

Allowing for some attributes to be ignored as irrelevant, we calculated the size of the corresponding hypothesis space as $|H| = 3^n$. If we strengthen the bias by insisting that every single attribute must be involved, we will reduce the size of the hypothesis space to $|H| = 2^n$. This is because every attribute in the class-describing expression is either *true* or *false*, whereas in the previous case, a third possibility ("ignored") was permitted.

Many other biases can be considered, some limiting the number of terms in the conjunction, others preferring a disjunction or some pre-specified combination of conjunctions and disjunctions, or imposing yet another constraint. What matters, in our context, is that each bias is likely to result in a different size of the hypothesis space. And, as we now know, this size affects learnability.

No Learning Without Bias! In the absence of *any* bias, allowing for *any* Boolean function, $\ln |H|$ grows exponentially in the number of attributes, $\ln |H| = 2^n \ln 2$, which means that this most general form of a classifier is *not PAC-learnable*. The explanation is simple. The unconstrained hypothesis space is so vast as to give rise to the danger that one of the classifiers will only accidentally label correctly the entire training set, while performing poorly in the instance space as a whole. The induced classifier then cannot be trusted. It is in *this* sense that we say, with a grain of salt, that "there is no learning without a bias."

The thing to remember is that any machine-learning undertaking can only succeed if the engineer constrains the hypothesis space by some meaningful bias. It stands to reason, however, that this bias should not be misleading. The reader will recall that our analysis assumed that the hypothesis space *does* contain the solution, and that the examples are noise-free.[3]

Preference for Simplicity Often, we can to choose from two or more biases. For instance, suppose the concept to be learned is described by a conjunction of attribute values in which each attribute has to be included. A weaker bias that permits the absence of some attributes from the conjunctions includes also the case where no such absence occurs (*zero* attributes are absent). The correct classifier therefore exists in both of these hypothesis spaces. In the former, we have $\ln |H| = n \ln 2$; and in the latter, $\ln |H| = n \ln 3$, which is bigger. We thus have two correct biases, each implying a hypothesis space of a different size. Which of them to prefer?

The reader already knows the answer. In Inequality 7.5, the number of training examples needed for successful learning depends on $\ln |H|$. A lower value of this term indicates that fewer examples are needed, and the engineer therefore chooses the bias whose hypothesis space is smaller. Again, this assumes that both hypothesis spaces contain the solution.

Occam's Razor Scientists have relied on this maxim for centuries: in a situation where two hypotheses appear to explain a phenomenon, the simpler one stands a higher chance of being correct. The fact that this principle, *Occam's Razor*, has been named after a scholastic theologian testifies that the rule pre-dates modern science by centuries. The word *razor* reminds us that, when formulating a hypothesis, we better slice away what is redundant.

That mathematicians have now been able to prove that this principle is valid at least in machine learning, and that they even quantified it, is a remarkable achievement.

Irrelevant and Redundant Attributes In the types of classes investigated in Sect. 7.2, the lower bound on the necessary number of examples, m, depended on the number of attributes, n. For instance, in the case of conjunctions of attribute values, we now know that $\ln |H| = n \ln 3$; and the number of examples needed to satisfy given (ϵ, δ)-requirements grows linearly in n.

The same result instructs us about learnability in the presence of irrelevant or redundant attributes. Including such attributes in the example description increases n, which means larger hypothesis space; and the larger the hypothesis space, the greater the size of the training set needed to satisfy the (ϵ, δ)-requirements. The

[3]Analysis of a situation where these requirements are not satisfied is mathematically more complicated and is outside the scope of this book.

lesson is clear: whenever we have a chance to identify (and remove) the less-then-useful attributes, we better do so.

These considerations also explain why it is so difficult to induce a good classifier in domains where only a tiny percentage of attributes carry the relevant information—as, for example, in automated text categorization.

7.3.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How do you interpret the statement, "there is no learning without a bias"? Conversely, under what circumstances will the bias tend to hurt the learning algorithm's chances of success?
- Explain the meaning of the term *Occam's Razor*. In what way did mathematicians put solid grounds under this philosophical principle?
- What does Inequality 7.5 tell us about the impact of irrelevant and redundant attributes?

7.4 VC-Dimension and Learnability

So far, we have focused on domains where the number of hypotheses is finite, which happens when all attributes are discrete. When at least one attribute is continuous-valued, the size of the hypothesis space is infinite. If we then need to compare learnability of two classes of classifiers, the formulas from the previous sections will be useless because the infinite $|H|$ will in both cases lead to infinite m.

This said, the reader will agree that some classes of classifiers (say, polynomial) are *more flexible* than others (say, linear), and that this flexibility should somehow affect learnability. This, indeed, is the case.

Shattered Training Set Consider the three points in the two-dimensional space depicted in the left part of Fig. 7.1. No matter how we distribute the positive and negative labels among them, there always exists a linear classifier that separates the two classes (some classifiers are shown in the same picture). We say that this particular set of examples is *shattered* by linear classifiers.

The case of the three points on the right of the same picture is different. These points find themselves all on the same line. If we label the one in the middle as positive and the other two as negative, no linear classifier will ever succeed in separating the two classes. We say that this particular set of examples is *not shattered* by the linear classifier.

Different classes of classifiers will differ in their ability to shatter given examples. For instance, a parabola (a special case of a polynomial function) will shatter the three aligned points on the right of Fig. 7.1; it will even shatter four points that do

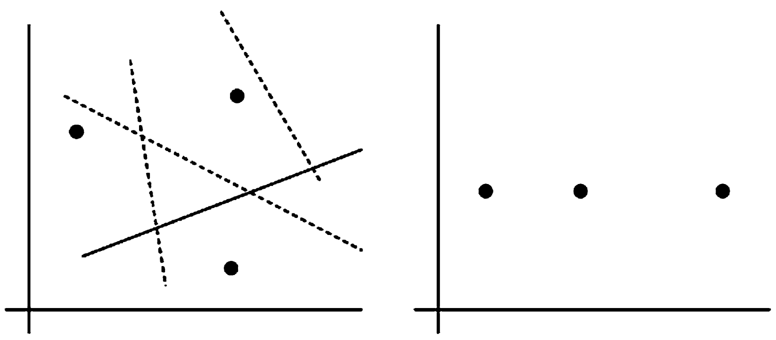


Fig. 7.1 The set of the three points on the left is *shattered* by a linear classifier. The set of the three points on the fight is *not shattered* by a linear classifier because no straight line can separate the point in the middle from the remaining two

not lie on the same line. Other classes of classifiers, such as high-order polynomials, will shatter any realistically sized set of examples.

Vapnik-Chervonenkis Dimension Each type of classifiers has its own Vapnik-Chervonenkis dimension (*VC-dimension*). This dimension is defined as the size of the largest set of examples shattered by the given classifier class.

We have seen that, in a two-dimensional space, a linear classifier fails to shatter three points that all lie on the same line, but that the linear classifier *does* shatter them if they do *not* lie on the same line. At the same time, four points, no matter how we arrange them in a plane, can always be labeled in a way that makes it impossible to find a separating linear function. Since the definition says, "the largest set of examples shattered by this class," we conclude that the VC-dimension of a linear classifier in the two-dimensional space is $VC_L = 3$.

The point to remember is that the value of the VC-dimension reflects the geometrical properties of the given type of classifier in a given instance space.

Learnability in Continuous Domains The concept of VC-dimension makes it possible to address learnability in continuous domains. Let us give here a slightly modified version of a famous theorem, omitting certain technicalities that are for our specific needs irrelevant:

Suppose that a certain classifier class, H has a finite VC-dimension, d. Then, error rate lower than ϵ can be achieved with confidence $1 - \delta$ if the target class is identical with some hypothesis $h \in H$, and if the number of the training examples, m, satisfies the following inequality:

$$m \geq \max(\frac{4}{\epsilon} \log \frac{2}{\delta}, \frac{8d}{\epsilon} \log \frac{13}{\epsilon}) \tag{7.9}$$

The lower bound on m is thus either $(\frac{4}{\epsilon} \log \frac{2}{\delta})$ or $(\frac{8d}{\epsilon} \log \frac{13}{\epsilon})$, whichever is greater.

Table 7.2 VC-dimensions for some classifier classes in R^n

Hypothesis class	VC-dimension
Hyperplane	$n+1$
Hypersphere	$n+2$
Quadratic	$\frac{(n+1)(n+2)}{2}$
r-order polynomial	$\binom{n+r}{r}$

In the worst-case analysis, the engineer can trust any classifier that correctly classifies the entire training set of m examples, regardless of the algorithm that has induced the classifier.

Note that the number of examples necessary for PAC learning grows linearly in the VC-dimension. This, however, is no reason to rejoice; the next paragraph will argue that VC-dimensions of many realistic classifiers have a way of growing very fast with the number of attributes.

Some Example VC-Dimensions Table 7.2 lists the VC-dimensions of some linear and polynomial classifiers. Note the trade-off: more complex classes of classifiers are more likely to contain the correct solution, but the number of training examples needed for success increases so dramatically that a classifier from such classes are not learnable in domains with realistic numbers of attributes.

Indeed, in the case of higher-order polynomials, the demands on the training-set size are all but prohibitive. For instance, the VC-dimension of a second-order polynomial in 100 dimensions is as follows:

$$d = \frac{102 \cdot 101}{2 \cdot 1} = 5,050$$

This is much greater than the VC-dimension of a linear classifier, but perhaps still acceptable (note that this value is sufficiently high to make the first term in Inequality 7.9 small enough to be ignored). But if we increase the polynomial's order to $r = 4$, the VC-dimension will become all but prohibitive:

$$d = \frac{104 \cdot 103 \cdot 102 \cdot 101}{4 \cdot 3 \cdot 2 \cdot 1} = 4,598,126$$

Polynomials of higher order thus should be avoided.

VC-Dimensions of Other Classifiers The high values of VC-dimension are the main reason why polynomial classifiers are almost never used; other classification paradigms are from this perspective much better. Mathematicians have been able to find the solutions for many different types of decision trees and neural networks. This chapter wants to avoid excessive mathematics. One illustration will be enough for the reader to develop an educated opinion.

Consider a multilayer perceptron whose neurons use the ReLU transfer function. Let L be the number of the network's layers, and let W be the number of trainable weights. Finally, let there be a constant, C, such that $W > CL > C^2$. Then the VC-dimension of the MLP is estimated by the following formula:

$$d = CWL \log W \tag{7.10}$$

This is smaller than the VC-dimension of higher-order polynomial classifiers.

General Rule of Thumb Mathematical analysis of the neural networks that are now used in the popular *deep learning* approaches (see Chap. 16) is not simple, but the message conveyed by the formula from the previous paragraph is loud enough:

> Whatever the classifier's architecture, whatever its sophistication, the number of the training examples should always exceed the number of trainable parameters, perhaps even by an order of magnitude.

Word of Caution Just as in the discrete-world Inequality 7.5, the solution reached in the continuous domains (Inequality 7.9) is the result of a worst-case analysis that relied on serious simplifying assumptions. The formula therefore should *not* be interpreted as telling us how many examples are needed in any concrete application. Its primary motivation is to help us compare alternative machine-learning paradigms such as polynomial classifiers versus neural networks.

Let us not forget that in realistic applications, examples are not labeled arbitrarily. Since the examples of the same class are somehow similar to each other, they tend to be clustered together.

7.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How does the number of examples that are necessary to satisfy the (ϵ, δ)-requirements grow with the increasing VC-dimension, d?
- What is the VC-dimension of a linear classifier in a continuous two-dimensional space?
- What is VC-dimension of an n-dimensional polynomial of the r-th order? What does its high value tell us about the utility of polynomial classifiers in realistic applications?

7.5 Summary and Historical Remarks

- Computational Learning Theory has established limits on the number of the training examples needed for successful classifier induction. These limits depend on two fundamental parameters: the classifier's requested error rate, ϵ, and the upper bound, δ, on the probability that m training examples will succeed in eliminating all classifiers with error rate greater than ϵ.

- If all attributes are discrete, the minimum number of examples needed to satisfy the (ϵ, δ)-requirements is determined by the size, $|H|$, of the given hypothesis space. Here is the classical formula:

$$m > \frac{1}{\epsilon}(\ln |H| + \ln \frac{1}{\delta})$$

- In a domain where some attributes are continuous, the minimum number of examples needed to satisfy given (ϵ, δ)-requirements is determined by the so-called VC-dimension of the given type of classifier. Specifically, if this VC-dimension is d, then the minimum number of examples is as follows:

$$m \geq \max(\frac{4}{\epsilon} \log \frac{2}{\delta}, \frac{8d}{\epsilon} \log \frac{13}{\epsilon})$$

- The two inequalities have been found by worst-case analysis. In practical circumstances, much smaller training sets are usually sufficient for the induction of high-quality classifiers. The main benefit of these formulas is that they help us compare learnability in different machine-learning paradigms.
- VC-dimension of r-th order polynomials in n-dimensional spaces is

$$d = \binom{n+r}{r}$$

- VC-dimension of MLP's with L layers ad W trainable weights is

$$d = CWL \log W$$

  where C satisfies $W > CL > C^2$.
- Whatever the classifier, the number of training examples should exceed the number of the classifier's trainable parameters.

Historical Remarks The principles underlying the idea of PAC learning were proposed by Valiant (1984). The classical paper on what later came to be known as VC-dimension is Vapnik and Chervonenkis (1971); somewhat more elaborate version was later developed by Vapnik (1992). The idea to apply VC-dimension to learnability, and to the investigation of Occcam's Razor is due to Blumer et al. (1989). Tighter bounds were later found, for instance, by Shawe-Taylor et al. (1993). VC-dimensions of linear and polynomial classifiers can be derived from the results published by Cover (1965). Readers interested in learning more about the *Computational Learning Theory* will greatly benefit from the excellent (even if somewhat older) book by Kearns and Vazirani (1994). That book, however, does not have much to say about learnability of neural networks—Formula 7.10 is from Bartlett et al. (2019).

7.6 Exercises and Thought Experiments

The exercises are to solidify the acquired knowledge. The ambition of the suggested thought experiments is to let the reader see this chapter's ideas in a different light and, somewhat immodestly, to provoke his or her independent thinking.

7.6.1 Exercises

1. Suppose that the instance space is defined by the attributes used in the "pies" domain from Chap. 1. Determine the size of the hypothesis space if the classifier is to be a conjunction of attribute values. Consider both cases: the one that assumes that some attributes might be ignored as irrelevant (or redundant), and the one that insists that all attributes must take part in the conjunction.
2. Return to the case of conjunctions of Boolean attributes from Sect. 7.2. How many more examples will have to be used (in the worst-case analysis) if we change the required error rate from $\epsilon = 0.2$ to $\epsilon = 0.05$? Conversely, how will the size of the necessary training set be affected if we change the value of δ?
3. Again, consider the case where all attributes are Boolean, and the classifier has the form of a conjunction of attribute values. What is the size of the hypothesis space if the conjunction is required to involve exactly three attributes? For instance, here is one conjunction from this class:

   `att1 = ` *`true`* `AND attr2 = ` *`false`* `AND att3 = ` *`false`*
4. Consider a domain with $n = 20$ continuous-valued attributes. Calculate the VC-dimension of a classifier that has the form of a quadratic function; compare this result with that for a third-order polynomial.
   Next, suppose that the engineer has realized that half of the attributes are irrelevant. Their removal will result in smaller dimensionality, $n = 10$. How will this reduction affect the VC-dimensions of the two classifiers?
5. Compare the *PAC-learnability* of the Boolean function involving 8 attributes with the *PAC-learnability* of a quadratic classifier in a domain with 4 numeric attributes

7.6.2 Give It Some Thought

1. A certain role in Inequality 7.5 is played by δ, a term that quantifies the probability that a successful classifier will be induced from the given training set. Under what conditions can the impact of δ be neglected?
2. From the perspective of *PAC-learnability*, is there a difference between irrelevant and redundant attributes?

3. We have seen that a classifier is often not *PAC-learnable* in the absence of bias. The right bias, however, many not be known. Suggest a learning procedure that would induce a classifier in the form of a Boolean expression in this case. (Hint: consider two or more alternative biases and suggest a strategy to evaluate them experimentally.)
4. In the past, some machine-learning scientists studied the possibilities of converting continuous attributes into discrete ones by the so-called *discretization*. By this they meant dividing an attribute's domain into intervals, and then treating each interval as a Boolean attribute that is *true* if the numeric value falls into interval and *false* otherwise.

   Suppose you are considering two ways of dividing the interval [0, 100]. The first consists of two sub-intervals, [0, 50], [51, 100], and the second consists of ten equally sized sub-intervals: [0, 10], . . . [91, 100]. Discuss the advantages and disadvantages of the two options from the perspective of *PAC-learnability*.

Chapter 8
Experience from Historical Applications

You will not become a machine-learning expert by just mastering a handful of algorithms—far from it. The world lurking behind the classroom toy domains has a way of complicating things, frustrating the engineer with unexpected hurdles, and challenging everybody's notion of what exactly the induced classifier is supposed to do and why. Just as everywhere in the world of technology, a healthy dose of creativity is indispensable.

Practical experience helps, too, and it does not have to be your own. You can just as well benefit from the positive and negative experience of those who experimented before you. This is the primary goal of this chapter. Building on a few old applications, it brings to light issues encountered in realistic applications, points out practical ways of dealing with them, and shows that the ambitious on machine learning cannot be reduced to low error rates. The reader may appreciate getting some sense of the questions that this discipline struggled with in its infancy.

8.1 Medical Diagnosis

A physician seeking to figure out the cause of her patient's complaints reminds us of a classifier: based on the available attributes (the patient's symptoms and laboratory tests), she suggests a diagnosis, the class label. At the dawn of machine learning, in the 1980s, many pioneers saw medical diagnosis as a natural target.

Machine-Learning Set Against Physicians The optimism was fed by early studies such as the one whose results are summarized in Table 8.1. Here, the testing-set accuracies achieved by Bayesian classifiers and decision trees are compared to those of physicians working with the same data. Each example represented a patient described by an attribute vector. The four domains differ in the number of classes as well as in the difficulty of the task at hand (e.g., noise in the data and reliability of the available information).

Table 8.1 Classification accuracy of two classifiers is compared to the performance of physicians in four medical domains

	Bayesian classifier	Decision tree	General practitioner
Primary tumor	0.49	0.44	0.42
Breast cancer	0.78	0.77	0.64
Thyroid	0.70	0.73	0.64
Rheumatology	0.67	0.61	0.56

The table suggests that it may indeed be possible to induce classifiers capable of competing with humans. Indeed, machine learning seems even to outperform them. In the multi-class `primary tumor` domain, the margin is unimpressive, but in the other domains, the classifiers seem to be clear winners. It is only a pity that we are not given more details such as standard deviations. Chapters 12 and 13 will have more to say about the methods to assess classification performance and its statistical reliability. In the 1980s, the "proof of concept" was far more important than sound scientific evaluations that become the norm much later.

Are the Results Encouraging? The numbers seem convincing, but let us not get overexcited. The first question to ask is whether the comparison was fair. Since the examples were described by attributes, the data available to machine learning could hardly be the same as those used by the physician who probably relied also on subjective information not available to the machine. The fact that the human enjoyed this advantage makes the machine's accomplishment all the more impressive.

On the other hand, the authors of the study admitted that the participating physicians were not specialists; machine learning thus only outperformed general practitioners. Let us also remind ourselves that the study was conducted almost two generations ago when laboratory tests were then far less sophisticated than now. Today, the data would be different and so would be the results of analogous comparison. Who would benefit more from modern diagnostic tools: the human or the machine? Hard to know.

Still, the results were encouraging and the hopes they inspired were justified.

Need for Explanations Diagnosis is not enough. A patient will hardly agree with a major surgery if the only argument in its support is, "this is what the machine says." The corroboration that "the machine's accuracy is on average 3% above that of a physician" is still unconvincing.

A reasonable person will ask why the surgery is preferable to conservative treatment. In the domains from Table 8.1, the doctor understands the evidence behind the diagnosis and has a good idea of alternative treatments. By contrast, Bayesian classifiers are virtually incapable of explanations. The tests in the decision tree do offer some clues, but these tend to be less than clear without expert interpretation.

Need to Quantify Confidence There is another problem to consider. A decision tree only decides that, "the example belongs to class 7." The physician and the patient want more. Is the recommended class guaranteed or is it just a bit more likely than some other class? If so, which are the competing classes?

Many application domains expect the computer to quantify its certainty about the returned class. For instance, it would be good to learn that, "the example belongs to C_1 with probability 0.8, and to C_5 with probability 0.7." Such quantification is provided by Bayesian classifiers, and similar information can be obtained also from neural networks. Both of these approaches, however, are bad at explaining.

Cultural Barriers One of the reasons these results failed to inspire more followers was poor strategy. Early enthusiasts just did not realize that medical doctors would hardly appreciate the prospect of being replaced by computers. They were less than excited about research that bragged about hoping to take away their jobs and prestige.

And yet it was all a misunderstanding. Later experience taught us that, in domains of this kind, machine learning is only to offer advice, the final decision remaining the responsibility of the physician, even in legal terms. And the advice can be more than useful. Machine-learning software can alert the doctor to previously unsuspected problems beyond the immediate diagnosis, and it can recommend additional laboratory tests.

8.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- In view of what you read in this section, why do you think it was impossible to say that the classifiers outperformed human experts?
- Apart from classification accuracy, what else does medical diagnosis need?
- Discuss the general limitations of machine-based diagnosis. Suggest a reasonable application of machine learning in medical domains.

8.2 Character Recognition

Another famous application area from early days of machine learning was correct identification of characters, hand-written or printed. Application possibilities are legion: automated pre-processing of forms, software for converting a text scribbled on a notepad into a file for a text editor, book scanning, and newspaper digitization, to mention but a few. Not so long ago, this was judged so ambitious as to be almost unrealistic; today, it is commonplace. Let us take a look at some of the lessons learned from this task.

Goal To describe a hand-written character in a way that allows its recognition by a computer is not easy. Just try to explain, in plain English, how to distinguish digit “5” from “6,” or what constitutes the difference between hand-written “t” and “l.” The difficulty will surprise you. And even if you succeed, converting the plain-English explanation to a computer program is harder still.

This is where machine learning can help. If we prepare a collection of pre-classified training examples and describe them by well-chosen attributes, then surely some of the techniques from the previous chapters will induce the classifier. The idea is so simple as to appear almost trivial—and in a sense this is indeed quite easy, provided that you have a good grasp of certain trade-offs.

Attribute Vectors Before starting the project, the engineer has to choose the attributes to describe the examples. Figure 8.1 illustrates one popular approach. Once the rectangular region with the digit has been identified, the next step divides it into, say, $6 \times 4 = 64$ equally sized fields, each represented by a continuous-valued attribute specifying the field’s mean intensity of reflected light (the more ink, the less light). Typically, 0 will stand for white and 255 will stand for black (it can be the other way round), with the numbers between the two extremes grading the shade of gray. Nowadays, it is possible to download benchmark domains with hundreds of thousands of hand-written characters. Most of these test-beds describe the examples in the manner just explained.

Less relevant from the machine-learning perspective is the need to identify the rectangle that fits the character. This is done by computer-vision software. Let us also mention that the classifier usually has to recognize not just one isolated character but rather “read” a text consisting of many such characters. This, however, is not much of a complication. As long as the individual characters can be isolated, the same way of describing them by attribute vectors can be employed.

Choosing the Level of Detail There is no reason why the grid should consist of the 64 attributes from Fig. 8.1. One can argue that this representation is so crude as to lose critical aspects of the character’s shape. We may decide to divide the rectangle

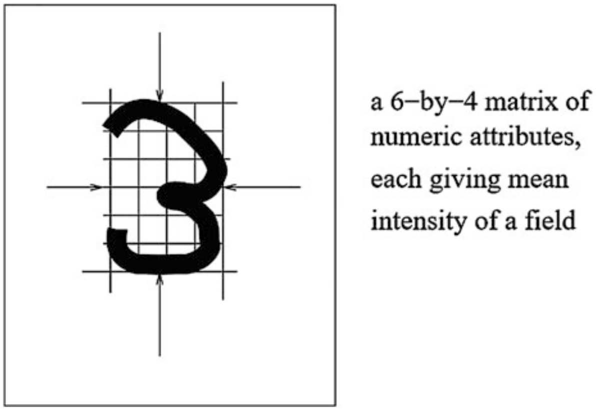

a 6–by–4 matrix of numeric attributes, each giving mean intensity of a field

Fig. 8.1 A simple way of converting an image of a hand-written character into a vector of 64 continuous attributes, each specifying the mean intensity of one field

into smaller fields, say, 40×60. The attribute vectors will then provide more detail—but then, computational learning theory is telling us that, with attribute vectors this long, many more training examples are needed. If we go the other way, dividing the grid to, say, 2×3, thus reducing the danger of overfitting, so much information is lost that nothing much can be learned. The engineer has to find the ideal middle ground.

Choosing the Classifier Now that we know how the training examples are going to be described, we can proceed to the next important questions: which machine-learning paradigm to choose, what induction technique to employ, and which aspects of the given task should be considered?

In the example description from Fig. 8.1, only a small percentage of attributes are likely to be irrelevant or redundant; this is therefore not an issue to worry about (unless the number of attributes is increased beyond those shown in Fig. 8.1). Also important is the fact that we do not know whether the classes are linearly separable—they *may* be separable if the number of attributes is high; in that event, however, the training set will have to be very large, perhaps much larger than the one at our disposal. Finally, the classifiers need not offer explanations. If the intention is to read, and pass to a text editor, a long text, the user will hardly care to know the exact reason why a concrete character was a “P” and not a “D.”

Based on these observations, the simple and easy-to-implement k-NN classifier seems a good choice.[1] The engineer may be worried by the computational costs incurred in a domain with hundreds of thousands of examples, but these may be acceptable if the number of attributes is reasonable. Actually, computational costs will still be dominated by the need to isolate the individual characters and to express them by attribute vectors. As long as these costs exceed those of classification (and they usually do), the classifier is affordable.

Until recently, the nearest-neighbor paradigm was indeed a common choice, often exhibiting error rates below 2%. In some really illegible hand-writings, the error rate is higher. But then, even humans may find them difficult to read.

Many Classes In a domain where all characters are capitalized, the induced classifier has to discern 10 digits and 26 letters, which means 36 classes. If both lowercase and uppercase letters are allowed, the total increases to $10 + 2 \times 26 = 62$ classes, and to these we may have to add special characters such as “?,” “!,” or “$.” Having to learn so many classes is not without consequences.

The most immediate concern is how to assess the practical benefits of the induced product. Error rate alone is not enough. Thus the performance of a classifier that correctly identifies 98% characters may appear acceptable; what this fails to tell us, though, is how the errors are distributed. Typically, some characters will be correctly identified most of the time, while others will be difficult. For instance, certain pairs of similar characters tend to be mutually confused. The practically minded engineer

[1] In Chap. 16, we will learn that the modern approach of deep learning may be better.

wants to know *which* pairs to focus on, perhaps to mitigate the problem by providing additional training examples of the harder classes.

Moreover, some letters are less common than others. Experience shows that these rarer classes tend to get "overlooked" by machine-learning algorithms unless special precautions have been taken. We will return to this issue in Sect. 11.2.

Rejecting An Example Certain hand-written characters are plainly ambiguous, and yet the k-NN classifier always finds a nearest neighbor and then simply returns its class, no matter how arbitrary it is.

In many domains, getting the wrong class can be more expensive than not knowing the class at all. Thus in an automated reader of postal codes, an incorrectly read digit can send the packet to a wrong destination, causing great delays. If the classifier does not give *any* answer, a human employee can do the reading, and the costs of manual processing may be lower than those caused wrong address.

The classifier should thus be implemented in a way that allows it to refuse to classify if the evidence behind the winning class is weak. The simplest way of doing so in the k-NN classifier is to set a minimum margin for the number of votes supporting the winner. For instance, if the winning class in a 7-NN classifier receives 4 votes, and the losing class 3 votes, the evidence in favor of the winner may be deemed insufficient and the classifier may be instructed to reject the example.

Something similar can be done also in other paradigms such as the Bayesian classifiers or neural networks. The classifier simply compares the probabilities (or output signals) of the two most likely classes and rejects the example if the difference fails to exceed a predefined threshold.

Error Rate Versus Rejection Rate A classifier that rejects ambiguous examples will exhibit low error rate. On the other hand, excessive reluctance to classify is impractical. What if *all* examples are rejected? The error rate is zero, but the classifier is useless.

The engineer has to consider the trade-off between rejection rate and error rate. Increasing the former is likely to reduce the latter; but care has to be taken not to overdo it.

8.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How can hand-written characters be described by attribute vectors?
- What aspects are considered (and why) when looking for the most appropriate machine-learning tool to be used?
- What is the consequence of the relatively high number of classes in this domain? Why does error rate fail to give the full picture?

- Why should the classifier refuse to classify certain examples? Discuss the trade-off between error rate and rejection rate. Comment on the interplay between performance and practical utility.

8.3 Oil-Spill Recognition

Figure 8.2 shows a radar image of the sea surface, taken by a satellite-born device. Against the grayish background, the reader can see several dark regions of the most varied characteristics: small or large, sharp or blurred, and of diverse shapes. The sharp elongated object close to the upper-right corner is oil illegally dumped by a tanker that chose to empty the residues of the bilges in the open sea rather than disposing of them at a terminal. This is against the law, and such oil spills are therefore of great interest to the Coast Guard and environmentalists alike.

Detecting the Violators In the mid-1990s, this inspired a project built around the following scenario. In certain areas of the sea surface (e.g., in the vicinity of major ports), satellites would take radar "snapshots" and forward them to ground stations. Experts would pore over the images, searching for illegal oil spills. Whenever they suspected one, an aircraft would be dispatched to the given location and verify the suspicion by an on-board spectrometer (unavailable to the satellite). The point was to collect evidence, perhaps even to identify the perpetrator.

Unfortunately, human experts are expensive—and not always available. They may be on holidays, on sick leave, or absent for some other reasons. Besides, the high number of images makes the work tedious and thus prone to mistakes. Someone suggested that a computer might automate the process. But how?

The picture from Fig. 8.2 has been selected out of many for its rare clarity. The oil spill it contains is so different from the other dark regions that even an untrained eye will recognize it. Even so, the reader will find it difficult to specify the spill's distinguishing features in a manner that can be used by a computer program. In more

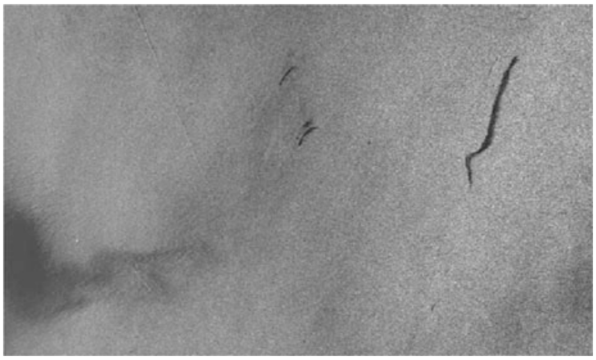


Fig. 8.2 A radar image of a sea surface. The "wiggly" elongated dark region in the upper-right corner represents environmental hazard: an oil spill

realistic objects, the task will be even harder. To hard-code the oil-spill recognition ability is indeed a challenge.

Automating the Process These difficulties led to the suggestion that perhaps machine learning might develop the recognition skills automatically, by induction from training examples.

(i) collect a set of radar images containing oil spills;
(ii) use image-processing software to find in these images dark regions;
(iii) ask an expert to label the oil spills as positive examples, and the other dark regions (the so-called "look-alikes") as negative examples;
(iv) describe all examples by attribute vectors, and let a machine-learning program induce the requisite classifier.

Attributes and Class Labels State-of-the-art image-processing techniques easily discover dark regions in an image. For the needs of machine learning, these have to be described by attributes likely to distinguish spills from look-alikes. The description should be unaffected by each object's size and orientation.

The attributes used here include the region's mean intensity, average edge-gradient (sharpness of the edges), ratio between the minor axis and major axis, variance of background intensity, variance of the edge gradient, and others. All in all, more than forty such attributes were selected in a rather *ad hoc* manner. Which of them would really be useful was hard to tell; experts were unable to reach consensus about the individual attributes' relevance. The final choice was thus left to the machine-learning software.

Labeling the training examples with classes was not any easier. The objects in Fig. 8.2 were easy to categorize; in other images, they were often ambiguous. On many occasions, the expert could only say that, "yes, this looks like an oil spill" or "I rather doubt this is what we're looking for." The correctness of the selected class was thus uncertain. In other words, the data were plagued by class-label noise.

Choosing the Classifier The examples were described by more than forty attributes. Among these, some, perhaps most, were likely to be irrelevant or redundant. To remove them, the scientists first induced a decision tree and then removed all attributes that the tree did not test. This made sense: the choice of the attributes to be included in the tree is guided by information gain, which is low in the case of irrelevant attributes. Also redundant attributes could thus be to a great extent eliminated.

When a k-NN classifier was applied to examples described by the surviving attributes, classification performance was even better than that of the decision trees.

Different Costs of Errors When evaluating the classifier's performance, one had to keep in mind that each type of error carried different costs. A false positive would result in an aircraft being unnecessarily dispatched, which meant wasted resources as well as "moral" costs—frequent failures could undermine the users' confidence. On the other hand, a false negative meant failure to detect an environmental hazard whose consequences (financial, environmental, and political) could not easily be expressed numerically. For these reasons, it was all but impossible to compare the costs of the two types of error: false positives versus false negatives.

Experiments indicated that most of the errors were of the false positive kind (false alarms), whereas false negatives were relatively rare. Was this good, or did it signal the need to modify the classifier in a way that might balance these errors?

The question could not be answered in isolation from immediate practical needs. It turned out that financial constraints might force the user occasionally to accept the risk of environmental hazard, simply because the budget could no longer tolerate false alarms. The user thus needed a mechanism to reduce the frequency of false positives even at the cost of an increased number of undetected oil spills (false negatives). The situation could change in more prosperous times when the user, unwilling to miss an oil spill, would accept false positives if this reduced the occurrence of false negatives.

Leaning Toward a Certain Class To address these trade-offs, the user needed a mechanism to adjust the classifier's behavior toward either type of error.

As already mentioned, this project relied on the k-NN classifier where this requirement is easy to address: the trick is to manipulate the number of votes needed for the winning class. Suppose the 7-NN classifier is used. The number of false positives can be reduced if we instruct the classifier to label as positive only those examples where, say, at least 5 of the nearest neighbors are positive (rather than the plain majority, which would be 4). Any example that fails to meet this condition will be deemed negative. Conversely, if we want to lower the frequency of false negatives, we tell the classifier to return the positive label whenever, say, at least 3 of the nearest neighbors are positive.

8.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How did the engineers identify redundant and irrelevant attributes?
- What can be said about the reliability of the class labels in the training set? What does it mean for the classifier's expected performance?
- Discuss the costs of false positive and false negative examples. Can the costs be assessed by the same quantities? Why does the user need a mechanism to reduce one type of error even at the price of increasing the other?
- Explain the mechanism that enables the k-NN classifier to increase or reduce either of the two errors.

8.4 Sleep Classification

Throughout the night, we go through different sleep stages such as deep, shallow, or rapid-eye movements (REMs) when we dream. To identify these stages in the subject, advanced instrumentation is used: an electrooculogram to record eye movements, an electromyogram to record muscle contractions, and contact electrodes attached to the scalp to record the brain's neural signals. Based on the readings of all these instruments, a medical expert determines what stage the sleeping subject is in at any given moment and can even draw a *hypnogram* such as the one shown in Fig. 8.3.

Note that the deepest sleep stage occurs here only three times during the 8-h sleep and that it usually does not last long. Note also the `move` stage. This occurs, for instance, when the subject turns from one side to another.

Why Is It Important? Correct identification of sleep states is important in medical practice. Consider the so-called *sudden infant death syndrome* (SIDS) where an infant dies without any apparent cause. To be able to start resuscitation, the hospital has to monitor the newborns suspected of being in danger. Uninterrupted monitoring is expensive; but then, SIDS almost always occurs during the REM stage. It is thus not necessary to watch the subject all the time, but only during this period of risk. A device to recognize the onset of REM would alert the nurse.

The hypnogram is a useful diagnostic tool because the distribution of the sleep stages during the night may indicate specific neural disorders such as epilepsy.

Why Machine Learning? To draw the hypnogram manually is a slow and tedious process, easily taking three to 5 h of a highly qualified expert's time. The expert is not always available and may be expensive, hence the suggestion to write a computer program to do the job automatically.

For this, the computer needs a description of each sleep stage. However, these descriptions are not available. Medical experts recognize sleep stages by studying

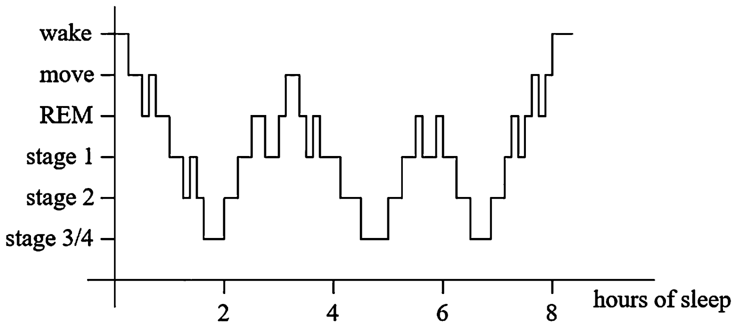


Fig. 8.3 An example hypnogram that records the sleep stages experienced by a subject during an 8-h sleep

EEC signals, relying on pattern-recognition skills gained after long training. The skills are too subjective to implement in a computer program.

The only solution thus was to induce a classifier from pre-classified data. To make it possible, scientists divided the 8-h sleep into 30-s periods, each constituting one training example. All in all, a few hours of sleep provided hundreds of training examples.

Attributes and Classes Again, the first task was to remove attributes suspected of being irrelevant or redundant. In the oil-spill domain, this removal increased the performance of the k-NN classifier. In sleep classification, another reason comes to the fore: the physician wants to minimize the number of measurement devices. Not only does their presence make the subject uncomfortable, but also they disturb the sleep and thus interfere with the results.

Class labels are here even less reliable than in the case of oil spills. The differences between "neighboring" (similar) sleep stages are so poorly defined that any two experts rarely agree on more than 70–80% class labels. The training data thus suffer from a lot of class-label noise, and the induced classifier cannot be expected to achieve low error rates.

Classifier's Performance The employed classifier combined decision trees with a neural network in a manner whose details are unimportant here. The classifier's accuracy on independent data indeed achieved those 70–80% observed in human experts, which means that performance limits have been reached.

Classification accuracy does not give the full picture, though. For one thing, the classifier correctly recognized some of the seven classes most of the time, while failing miserably on others. Particularly disappointing was the REM stage. Here, classification accuracy was in the range of 90–95%, which, on the surface, seemed impressive. However, closer inspection revealed that less than 10% of all examples represented REM; this means that comparable accuracy would be achieved by a classifier claiming that "there are no REM examples"—which of course would be useless.

Evidently, classical ways of measuring performance by error rates and accuracies are in domains of this kind misleading. More appropriate performance criteria will be introduced in Chap. 12.

Improving Performance by Post-processing The hypnogram's accuracy can be improved by post-processing. Several rules of thumb can be employed. For instance, the deepest sleep (`stage 3/4`) is unlikely to occur right after REM, and `stage 2` does not happen after `move`. Further on, the hypnogram can be "smoothed out" by the removal of any stage that lasts only one 30-s period.

The lesson is worth remembering. In domains where the examples are ordered in time, the classes of some examples may not be independent of those preceding or following them. Post-processing can then improve the classifier's performance.

Proper Use of the Classifier The original idea was to induce the classifier from examples obtained from a few subjects and then to use it to classify all future data. This proved unrealistic. Experience showed that no "universal classifier" of sleep could thus be obtained: a classifier induced from one person's data could not be used to draw a hypnogram for another person without serious loss in accuracy.[2]

This does not mean that machine learning is in this domain disqualified. Rather, the lesson is that users have to modify their expectations: instead of a universal solution, the users have to be content with a separate classifier for each subject; the expert's labors are still significantly reduced. The following scenario is indicated:

(i) The expert determines the class labels of a subset of the available examples (from a single subject), thus creating a training set.
(ii) From this training set, a classifier is induced.
(iii) The induced classifier is used to classify the remaining examples (of the same object), thus saving the expert's time.

8.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Why was it important to minimize the number of attributes? How could the relevant attributes be identified?
- What circumstances have be considered by an engineer seeking to evaluate the induced classifier's performance?
- In the hypnogram, the examples are ordered in time. How can this be exploited in improving the results?
- In what manner can machine learning reduce the burden imposed on someone who seeks to classify available data?

8.5 Brain–Computer Interface

The muscle-controlling commands are issued at specific locations of *motor cortex*, a relatively well-understood region of the cortex. Even the brain of totally paralyzed patients can often generate relevant signals, but the information does not reach the muscles. This observation inspired an ambitious idea: can the brain signals, detected by electrodes and properly processed, be used to move a cursor on a computer screen? A positive answer would help the paralyzed communicate with the outside world, even if the patient is unable to speak and even unable to move his pupils.

The exact nature of the motor commands is too complicated to be reduced to a mathematical formula. What we can do is record the signals, describe them by

[2]Perhaps a better set of attributes might help; the case study reported here did not attempt to do so.

attributes, and label them with concrete commands. This would create a training set from which to induce the classifier.

Training Examples The study considered only two classes: `left` and `right`. The training and testing examples were created by the following procedure. A subject was sitting in front of a computer monitor. On the desk in front of the subject was a plywood board with two buttons, one to be pressed by the left index finger and the other by the right index finger, according to the instructions displayed on the monitor.

The scenario followed the time line shown in Fig. 8.4. The contact electrodes attached to the scalp (see Fig. 8.5) recorded the intensity of the neural signals during a certain period of time just before the button was pressed. The signals from each electrode were represented by five numeric attributes, each giving the power of the signal over a time interval (a fraction of a second).

As indicated in the picture, only eleven electrodes were actually used. With five attributes per electrode, this meant 55 attributes. The training set contained a few hundred examples with both classes equally represented. Since it was always known which button was pressed, the training set was free of class-label noise. As for the attributes, many of them were suspected to be irrelevant.

Classifier, Its Performance, and Its Limitations Several machine-learning techniques were tried, including multilayer perceptrons and nearest-neighbor classifiers

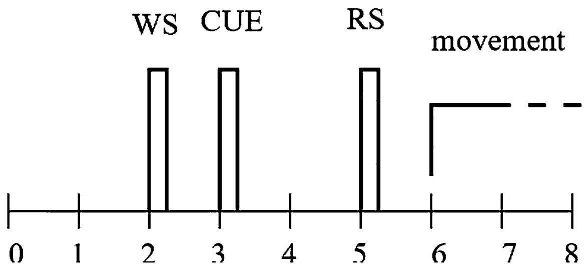


Fig. 8.4 After a "ready" signal (WS) comes the CUE ("left" vs. "right"). Once RS appears on the screen, the subject waits one second and then presses the button indicated by the CUE

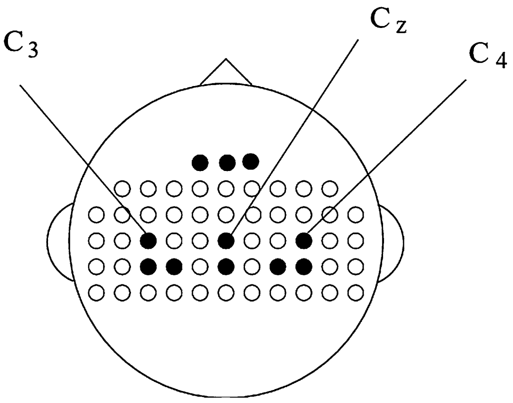


Fig. 8.5 Arrangement of electrodes on the subject's scalp. Only the highlighted electrodes were actually used

with attribute-value ranges normalized so as to avoid the scaling problems. Relevant attributes were selected by a decision tree, similarly as in the previously described domains.

Typical error rates of the induced classifiers on testing data were in the range of 25–30%, depending on the concrete subject. It is quite possible that higher accuracy could never be achieved: the information provided by the given attributes was probably insufficient.

Just as in the sleep classification domain, the classifier could only be applied to the subject from whose training data it was induced. All attempts to induce a "general classifier" (to be used on any future subject) failed: the error rates were so high as to make the classifications look random.

Evaluating the Classifier's Utility Just as in the previous domain, the error rate measured on independent data fails to provide the whole picture. The classifier's practical utility depends on the way it is used. The ultimate test is: does the classifier succeed in sending the cursor to the right place?

This was assessed by the following experiment. The subject was still sitting in front of a computer monitor, with electrodes attached to his or her scalp, but the board with the two buttons had been removed. The monitor showed two rectangles (see Fig. 8.6), one on the left and one on the right. The cursor was in the middle.

When instructed to move the cursor, the subject only *imagined* he was pushing the corresponding button. The electrodes recorded the neural signals thus generated and passed them to the classifier that decided where the cursor was to be sent (`left` or `right`). The cursor's movement was very slow so as to give the subject the opportunity to correct a wrong direction. The neural signals were sampled repeatedly, each sample becoming one example to be forwarded to the classifier.

Suppose the subject's intention was to move the cursor to the right. It could happen that the first sample was misclassified as `left`, sending the cursor in the wrong direction; but if the following samples were correctly classified as `right`, the cursor did, after all, land in the correct rectangle, even if only after what looked like the machine's hesitation.

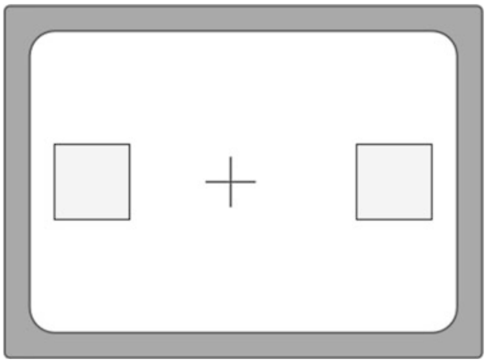


Fig. 8.6 The task is to move the cursor into either the left box or the right box, according to the subject's "thoughts"

Do We Need Minimum Error Rate? Under the scenario just described, an error rate of 25–30% is quite acceptable. Even when the cursor occasionally did start in the wrong direction (on account of some examples being misclassified), the mistake was corrected by later examples, and the cursor did reach the intended destination.

As a matter of fact, the cursor almost always landed where it was asked to go, even though it sometimes did so later than it should have. An impartial observer, uninformed about how the classifier was actually employed, rarely noticed any problem. The occasional back-and-forth movements of the classifier were perceived as a sign of indecisiveness on the part of the subject and not of the classifier's imperfection.

Only when the classifier's error rate dropped well below those 30% did the cursor miss its target convincingly enough to cause disappointment. Perhaps a better measure of the classifier's performance would in this case be the average time needed to reach the correct box.

An Honest Remark The classifier's performance differed from one person to another. In many subjects, the error rate was almost prohibitively high. Also the fact that the classifier had to be trained on the same subject on which it was to be used was perceived as a shortcoming. Further on, two classes are not enough. At the very least, also `up` and `down` would be needed. On these, experimental results were less convincing. In the end, other methods of communication with paralyzed patients came to be preferred. Though successful as a machine-learning exercise, the project did not meet the expectations of the medical sciences.

8.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Discuss the experience made in this application: a classifier induced from the data of one subject cannot be used to classify examples in another subject.
- Explain why the error rate of 30% was in this domain still deemed acceptable. What other methods of measuring performance, in this application, would you recommend?

8.6 Text Classification

Consider a large repository of perhaps a million on-line text documents. When presented with a query specifying a concrete topic, the system is to return all relevant documents. For this to be possible, the documents need to be *annotated*: labeled with the topics they represent. Since the annotation of millions of documents cannot be accomplished manually, machine learning has been used to automate the process.

Here is the idea. A group of experts select a small but representative subset of the documents, read them, and then label each document with the class specifying its topic. In this way they create a training set from which a classifier is induced. This classifier is then used to label the rest of the repository. If only a few thousands of examples are used for training, a lot of time-consuming work can be saved.

Attributes A common way to describe a text document is by the the words it contains. Each attribute represents one word, and its value gives the frequency of the word in the text. For instance, if in a document consisting of 983 words, the term `railroad` appears five times, its frequency is $5/983 = 0.005$.

General vocabulary being quite rich, an attribute vector of this kind will be very long. Since only a small percentage of the words will be found in a given document, the vast majority of the frequencies will be zero. Simple applications therefore prefer to work with much shorter vectors of, say, 1000 most common words.

Class Labels Labeling text documents is subjective. Consider the term `computer science`. A scientific paper dealing with algorithm analysis is surely a positive example; a paper that only mentions in passing that neural-network training is computationally expensive is less typical, and a brief article from a popular magazine may be totally irrelevant from the perspective of a scientifically minded reader—though fairly relevant for general readership.

One way to handle this situation is to "grade" the class labels. Previously, we considered for the given class only 1 or 0. Instead, we may use 2 if the document is definitely relevant, 1 if it is only somewhat relevant, and 0 if it is totally irrelevant. The users can then decide what level of relevance they request. They may interpret as positive all examples where the value is 2 or all examples that are relevant at least at the level of 1.

Observations The very long attribute vectors make induction expensive. Thus in the case of decision trees, where the program has to calculate information gain separately for each attribute, tens of thousands of attributes need to be investigated for each tree node, and these calculations will be costly if the training set is large.

True, in the case of k-NN or linear classifiers, induction is cheaper. Here, however, another difficulty comes to the fore: for any class, the vast majority of the attributes will be irrelevant; this, we already know, complicates learnability.

Multi-Label Examples In the previous sections, each example was labeled with one and only one class. Text classification is different in that each document typically belongs to two or more (sometimes many) classes at the same time.

The most common solution is to induce a separate classifier for each class. Whenever a future document's class is needed, its attribute vector is submitted to all classifiers in parallel; some of them will return "1" and others will return "0." This adds to computational expenses because of the need to induce hundreds of classifiers instead of just one.

Induction from multi-label examples causes some other problems, and it turns out that discussing all of them calls for the entire Chap. 14.

8.6.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How will you describe examples in a text-classification domain? Why are the attribute vectors so long? What difficulties are posed by these vectors?
- What did this section mean by "graded class labels"?
- What makes induction of text classifiers expensive?

8.7 Summary and Historical Remarks

- The number of examples to be used for learning varies. In some domains, examples are abundant, for instance, if they are automatically extracted from a database. In others, they are rare and expensive as in the oil-spill domain.
- Often, we have no idea which attributes really matter. In the oil-spill domain, the shape and other characteristics could be described by an almost unlimited number of attributes obtained by image-processing techniques. Many, perhaps most of these attributes are bound to be irrelevant or redundant.
- In some domains, critically important attributes are either not known or cannot be obtained at reasonable costs. Inevitably, machine learning has to work with those attributes that are available, even if the performance of the induced classifier is then limited.
- In the brain–computer interface, it was okay that the majority of the decisions were correct, just to make the cursor land in the correct rectangle in reasonable time. Hundred percent classification accuracy was not needed.
- In domains with more than two classes, error rate does not give the full picture of the classifier's behavior. Quite often, some classes are almost always perfectly recognized, while others pose problems. It is thus important to know not only the average performance but also the performance for each individual class.
- In medical diagnosis, low error rate is not enough. The user wants the decision to be explained and argued for.
- The costs of false positives can differ from the costs of false negatives. In the oil-spill domain, the respective costs could not easily be expressed in monetary terms, and it was all but impossible to compare them.
- In the oil-spill domain, the user needed some parameter that would tilt the classifications in ways that trade false positives for false negatives and the other way round.
- In the text-categorization domain, the attribute vectors can be very long, and most of the attribute values are zero. Besides, a typical example in this domain will belong to several classes at the same time.

Historical Remarks The results from Table 8.1 are from Kononenko et al. (1998) who discuss there an older project of theirs. The oil-spill project was reported by Kubat et al. (1998). The sleep classification task is addressed by Kubat et al. (1994), and the experience with using machine learning in brain–computer interface was published by Kubat et al. (1998). The character-recognition problem has been addressed for decades by the field of Computer Vision; the first major text systematically addressing the issue from the machine-learning perspective seems to be Mori et al. (1992). The text-classification task was first studied by Lewis and Gale (1994).

An explanatory remark is in place, here. The reader may have noticed that, of these applications, the author of this book participated in three. His motivation for including them was not to impress the reader with his incredible scholarship—far from it. The truth is, if you work on a project for some time, you become not only personally attached to it but also develop certain deeper understanding and appreciation of the problems you have encountered. This makes such project much more appropriate for instruction than projects that you have read about in the literature.

8.8 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The ambition of the suggested thought experiments is to let the reader see this chapter's ideas in a different light and, somewhat immodestly, to provoke his or her independent thinking.

8.8.1 Give It Some Thought

1. Suppose you are not certain about the correctness of some class labels in your training set. Would you recommend that these "unreliable" examples be removed? Under what circumstances? Do not forget that some of the pre-classified examples have to be set aside as testing examples to evaluate the induced classifier's performance.
2. Discuss the reasons why application-domain users may be reluctant to accept the machine-learning tools and results. Suggest ways to address their suspicions and concerns.
3. Section 8.3 mentioned a simple mechanism that enables the k-NN classifier to manipulate the two kinds of error (false negatives versus false positives). Suggest a similar mechanism for Bayesian classifiers and neural networks. How would you go about implementing such mechanism in a linear classifier?
4. In some of the domains from this chapter, it was necessary to identify irrelevant and/or redundant attributes. In these particular projects, decision trees were used.

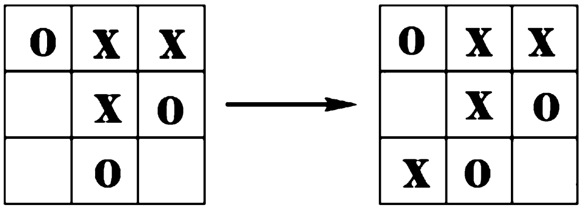


Fig. 8.7 In tic-tac-toe, the goal is to achieve three crosses (or circles) in a row or a column or on a diagonal

Suggest possibilities based on what you learned in earlier chapters. Discuss their advantages and disadvantages.

5. Suppose you want to implement a program to decide whether a given position in the tic-tac-toe game (see Fig. 8.7) is winning. What attributes would you use? How would you collect the training examples? What can you say about expected noise in the data thus collected? What classifier would you use? What difficulties are to be expected?

8.8.2 Computer Assignments

1. Some realistic data sets for machine-learning experimentation can be found on the website of the National Institute of Standards and Technology (NIST). Find this website, and then experiment with some of these domains.
2. Find a website dealing with the demography of the 50 states of the U.S. Identify an output variable that will be positive if the state exceeds the U.S. average and negative otherwise. Each state thus constitutes an example. Based on the information provided by the website, identify the attributes to describe them. From the data thus obtained, induce a classifier to predict the output variable's value.

Chapter 9
Voting Assemblies and Boosting

A popular way of dealing with difficult problems is to organize a brainstorming session in which experts from different fields share their knowledge, exchanging diverse points of view that complement each other in ways likely to inspire unexpected solutions. Something similar can be done in machine learning, as well. A group of classifiers is created, each of them somewhat different. When they vote about a class label, their "collective wisdom" often compensates for each individual's imperfections. This results in higher classification performance.

This chapter deals with mechanisms for the induction of sets of classifiers. The reasons behind the high performance of these *classifier assemblies* are explained on the simplest approach known as *bagging*. Building on this foundation, the text then proceeds to the more sophisticated *boosting* algorithms and their variations, including *random forests* and *stacking*.

9.1 Bagging

For simplicity, let us constrain ourselves to two-class domains where each example is either positive or negative. As always, the classifier is to be induced from a set of pre-classified training examples.

Underlying Principle The approach known as *bagging*[1] induces a group of classifiers, each from a different subset of the training data. When presented with an example to be classified, the classifiers are all applied in parallel, each offering an opinion about the example's class. A *master classifier* then decides which label got more votes.

[1]The name is an acronym: $\underline{b}$*ooststrap* $\underline{agg}$*regation*.

Table 9.1 Pseudo-code of *bagging*

```
Input: training set, $T$, and the user's choice of the baseline induction technique

1. Using random sampling with replacement, create from $T$ several training subsets,
   $T_1, T_2, \dots T_n$. Each subset consists of the same number of examples.
2. From each $T_i$, induce classifier $C_i$.
3. An example to be classified is submitted to all $C_i$'s in parallel, each of them suggesting the
   example's class.
4. A master classifier decides which of the two classes received more votes.
```

Assuming that each of the participating classifiers represents a somewhat different aspect of the recognition task, the group of classifiers (also known as a "voting assembly") is likely to outperform any individual.

Induction of the Voting Assembly The principle of *bagging* is summarized in Table 9.1. The idea is to take the original training set, T, and create from it a certain number of random training subsets, $T_1, \dots T_n$, each of the same size.

Once the subsets T_i have been created, a machine-learning algorithm induces from each of them a classifier, C_i. For this induction, any of the techniques from the previous chapters can be used. However, the baseline version of *bagging* assumes that the same technique (say, induction of decision trees with the same user-set parameters) is used on each T_i.

Bootstrapping Let us now explain how to create the training subsets, T_i. Each example from T has the same chance of being picked. Once it *has* been picked, it is copied to T_i, but still left in T so that it can later be selected again, with the same probability. This is what statisticians call "random selection with replacement." In their thought experiments, once a ball has been selected from an urn, it is put back (replaced) so that it can be chosen again. For a training set, T, consisting of N examples, the selection is repeated n times, thus generating n subsets. The process is known as *bootstrapping*.

An example can appear in T_i more than once and, conversely, some examples will not appear in T_i at all. Each T_i consists of N examples (where N is the size of the original training set, T), but different in each training subsets. Mathematicians have been able to show that, in this manner, about two-thirds of the examples in T will find their way to T_i.

Since no two training subsets are the same, each of the induced classifiers is focused on different aspects of the given problem, and each of them exhibits a somewhat different classification behavior.

Why It Works Figure 9.1 helps explain why the method helps reduce error rate. Three classifiers are considered. Each of them classifies correctly many examples but fails on others. If the errors are rare, there is a good chance that each of the three classifiers will err on different examples, which means that each example will

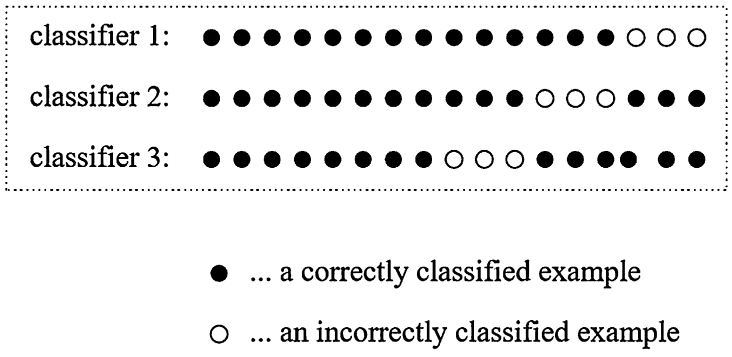

classifier 1:
classifier 2:
classifier 3:
● ... a correctly classified example
○ ... an incorrectly classified example

Fig. 9.1 Each of the three classifiers classifies the same 17 examples. Each classifier errs on three examples—different for each classifier. The picture shows how these errors are corrected by voting

be misclassified at most once. The class labels of the other two classifiers being correct, an individual's occasional mistake is corrected by the other classifiers.

Of course, this is only an ideal situation. Often, some examples will be misclassified by two out of the three classifiers, in which case the voting results in the wrong class. One can assume, however, that these errors will become less frequent if we increase the number of classifiers.

The point to remember is that *bagging* succeeds if each classifier tends to err on different examples.

Observations Experience shows that *bagging* improves classification performance if the error rates of the individual classifiers are low. With a sufficiently large number of classifiers, chances are high that some individuals' errors will be corrected by the other classifiers in a way indicated in Fig. 9.1.

This may not be the case when the error rates of the classifiers are high. In that event, the wrong class may be suggested by the majority, and voting will not help. The situation improves if a great many classifiers are used because then, by the law of large numbers, each aspect of the underlying recognition problem is likely to be represented. For training sets of realistic sizes, however, the number of classifiers used in *bagging* rarely exceeds 100.

Too Much Randomness The principle shown in Fig. 9.1 suggests that the classifiers perhaps should *not* be created at random; rather, they should be induced in a way that makes each of them err on different examples so that the classifiers complement each other as much as possible. Mechanisms built around this advice are the subject of the following sections.

9.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What makes us believe that a group of voting classifiers will outperform a single classifier? When will the scheme fail?
- How are the individual classifiers induced in the *bagging* approach?
- Explain the principle of *bootstrapping*.

9.2 Schapire's Boosting

Although *bagging* often leads to impressive results, it suffers from a serious shortcoming: the voting classifiers have all been induced independently of one another. A smarter approach will rely on a mechanism that makes the classifiers complement each other in the spirit of Fig. 9.1. For instance, the classifiers should be induced one at a time, each focusing on those training examples that have proved to be difficult for the previous classifiers. *Schapire's boosting* was invented with this idea in mind.

Three Mutually Complementing Classifiers Suppose that a random subset, $T_1 \in T$, of m training examples has been created. From this, the first classifier, C_1, is induced. Testing this classifier on the entire training set, T, will reveal that it misclassifies a certain number of examples.

Let us now create a second subset, $T_2 \in T$, consisting of m examples selected in a manner that ensures that the previously induced C_1 classifies correctly 50% of them, failing on the remaining 50%. Put another way, T_2 is so difficult for C_1 that the classifier will not outperform a flipped coin. From T_2, the second classifier, C_2, is induced.

Having been each induced from different examples, C_1 and C_2 will differ in the way they label some examples. To obtain a tie-breaker, a third training subset, T_3, is created exclusively from examples on which C_1 and C_2 disagree (again, there should be m of them). From T_3, the third classifier, C_3, is induced.

The whole procedure is summarized by the pseudo-code in Table 9.2. Ideally, each of the training sets, T_i, has the same size, m, so that each of the induced classifiers has the same authority. When an example is presented, a *master classifier* decides which class received more votes.

Recursive Implementation The principle can be implemented recursively; Figure 9.2 shows how. The triplet of classifiers obtained by the algorithm in Table 9.2 (in the dotted rectangle) is treated as a single classifier. In the next step, a new training subset, T_4, is created in a manner that ensures that the triplet's error rate on T_4 is 50%. From these, classifier C_4 is induced. Finally, training subset T_5 is

Table 9.2 Pseudo-code of *Schapire's boosting*

```
Input: training set, $T$, and the user's choice of the baseline induction technique

1. Create a random training subset, $T_1$, and induce from it classifier $C_1$.
2. Create a training subset $T_2$ in a manner that makes sure that $C_1$ scores 50% on it. From $T_2$,
   induce classifier $C_2$.
3. Create a training subset $T_3$ from examples on which $C_1$ and $C_2$ disagree. From $T_3$, induce
   classifier $C_3$.
4. For classification, let $C_1$, $C_2$, and $C_3$ vote.
```

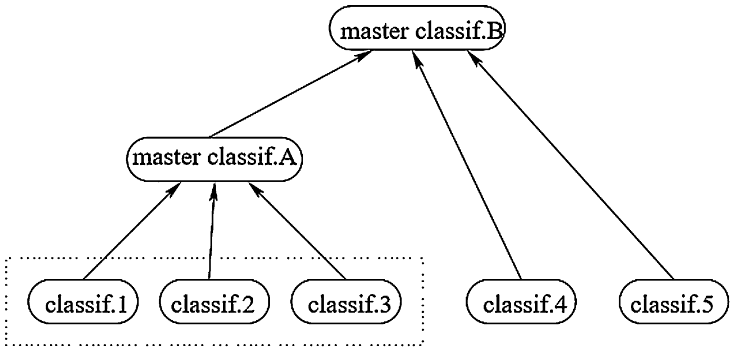


Fig. 9.2 Recursive application of *Schapire's boosting*. Master classifier A combines the votes of classifiers 1–3, and master classifier B combines the votes of master classifier A with those of classifiers 4 and 5

created from examples on which the triplet and C_4 differ; from these, classifier C_5 is induced. Figure 9.2 also shows the hierarchical organization of the voting.

Note the mechanism of the voting procedure. If classifiers 1 through 3 all return 0, master classifier A returns class 0, as well; however, this result can still be outvoted if both classifier 4 and classifier 5 return class 1. The whole structure thus may return 1 even if three out of the five participating classifiers return 0.

The total number of classifiers induced using this single level of recursion is $3 + 2 = 5$. The principle can be repeated, resulting in $5 + 2 = 7$ classifiers, and so on. Assuming N_R levels of recursion, the total number of participating classifiers is $2N_R + 3$.

Performance Considerations Suppose that each of the induced classifiers has error rate below a certain ϵ. It has been proved that the voting triplet's error rate is then less than $3\epsilon^2 - 2\epsilon^3$, which is always smaller than ϵ. For instance, if $\epsilon = 0.2$, then $3\epsilon^2 - 2\epsilon^3 = 2 \cdot 0.04 - 2 \cdot 0.008 = 0.104$. And if $\epsilon = 0.1$, then $3\epsilon^2 - 2\epsilon^3 = 0.03 - 0.002 = 0.028$.

Put another way, *Schapire's boosting* seems to guarantee an improvement over the performance of the individual classifiers. However, if the first voting triplet achieves $3\epsilon^2 - 2\epsilon^3 = 0.104$, it may be difficult to achieve an equally low error rate,

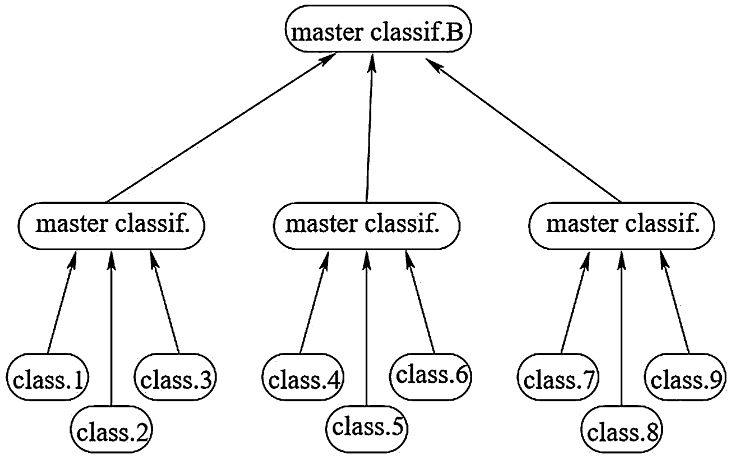


Fig. 9.3 Another possibility of recursive application of *Schapire's boosting*

0.104, with classifiers 4 and 5. One way to overcome this difficulty is to conceive each of them (classifier 4 and classifier 5) as a triplet in its own right, as in Fig. 9.3. The number of classifiers participating in N_R levels of recursion will be 3^{N_R}.

The thing to remember is that, at least in theory, each added level of recursion reduces the error rate. This seems to promise the possibility of reducing the error rate almost to zero. Practice, however, is different—for reasons explained in the next paragraph.

Limitations The main difficulty is to find the right examples to be included in each of the subsequent training subsets. These, we already know, have to satisfy certain criteria: to induce the second classifier, C_2, we need a training subset, T_2, that makes the previous classifier, C_1, classify correctly only 50% examples; and the third classifier is induced exclusively from examples on which C_1 and C_2 disagree.

This is easier said than done. If the entire training set consists of 100 examples, and the first classifier's error rate is 10%, then we have only 10 misclassified examples, to which 10 correctly classified examples are added to create T_2; this means that the size of T_2 will not exceed 20. Likewise, we may find it impossible to find m examples on which C_1 and C_2 give the opposite class labels.

Even if we do succeed in creating equally sized three training subsets satisfying our criteria, we may not be able to apply the principle recursively. For this, we would need an almost inexhaustible (and certainly not affordable) source of examples—a luxury hardly ever available.

9.2.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How does *Schapire's boosting* create the training subsets from which the individual classifiers are to be induced?
- Explain the two ways of implementing the principle recursively. How many classifiers are induced in each case if N_R levels of recursion are used? Discuss the voting mechanism.
- When compared to the error rate of the individual classifiers, what is the error rate of the final classification?
- Discuss the practical limitations of *Schapire's boosting*: the problem with finding enough examples.

9.3 Adaboost: Practical Version of Boosting

The main weakness of *bagging* is its randomness: the classifiers are induced from random data, independently of each other. In *Schapire's boosting*, the randomness is minimized, but there was another problem: in a realistic setting, it is often impossible to find the training examples that satisfy the stringent conditions.

These observations motivated *Adaboost* where the training subsets are chosen using a probabilistic distribution. This distribution is systematically modified in a way that helps focus on the gradually more difficult aspects of the given class.

General Idea Similarly to Schapire's approach, *Adaboost* creates the classifiers one by one, each from a different training subset whose composition depends on the behavior of the previous classifiers. There is a difference, though. Whereas *Schapire's boosting* selects the examples according to precisely defined conditions, *Adaboost* chooses them probabilistically. Each example has a certain chance of being drawn, the probabilities of all examples summing to 1. Examples that were repeatedly misclassified by previous classifiers get a higher chance of being included in the training subset for the next classifier.

Another difference is the number of classifiers. Unlike Schapire's triplets, *Adaboost* typically relies on a great number of classifiers. Further on, the final decision is not achieved by the plain voting used in *bagging* but rather by *weighted majority voting*.

Probabilistic Selections As mentioned, the training subsets, T_i, are created from the original set, T, using probabilities. Here is a simple way of doing so. Suppose the i-th example's chances of being drawn are specified as $p(\mathbf{x_i}) = 0.1$. A random-number generator is asked for a number between 0.0 and 1.0. If the returned number is from interval [0.0, 0.1], the example is selected; otherwise, it is not.

Table 9.3 Pseudo-code of *Adaboost*

```
Input: training set, $T$, of $m$ examples; the user's choice of the induction technique

1. Let $i = 1$. For each $\mathbf{x}_j \in T$, set $p_1(\mathbf{x}_j) = 1/m$.
2. Create subset $T_i$ consisting of $m$ examples randomly selected according to the given probabilities. From $T_i$, induce $C_i$.
3. Evaluate $C_i$ on each example, $\mathbf{x}_j \in T$.
   Let $e_i(\mathbf{x}_j) = 1$ if $C_i$ misclassified $\mathbf{x}_j$ and $e_i(\mathbf{x}_j) = 0$ otherwise.
     (i) Calculate $\epsilon_i = \sum_{j=1}^{m} p_i(\mathbf{x}_j)e_i(\mathbf{x}_j)$;
     (ii) Calculate $\beta_i = \epsilon_i/(1 - \epsilon_i)$.
4. Modify the probabilities of correctly classified examples by $p_{i+1}(\mathbf{x}_j) = p_i(\mathbf{x}_j) \cdot \beta_i$
5. Normalize the probabilities to ensure that $\sum_{j=1}^{m} p_{i+1}(\mathbf{x}_j) = 1$.
6. Unless a termination criterion has been met, set $i = i + 1$ and go to 2.
```

At the beginning, when the first training set, T_1, is being created, each example has the same chance: if T consists of m examples, then the probability of each example is $p = 1/m$. For each of the next subsets, T_i, the probabilities are modified according to the observed behavior of the previous classifier, C_{i-1}. The idea is to make sure that examples misclassified by the previous classifiers should get a higher chance of being included in T_i than those that were correctly classified. This will focus C_i on those aspects that the previous classifiers found difficult.

Modifying the Probabilities The first training subset, T_1, is created from examples that all had the same probability of being included in it: $p = 1/m$. After this, the probability of examples correctly classified by C_1 is reduced, and the probability of examples misclassified by C_1 is increased.

The way the probabilities are modified is shown in Table 9.3. First, *Adaboost* calculates the i-th classifier's overall error, ϵ_i, as the weighted sum of errors on the whole original training set: this is obtained simply by summing the probabilities of the misclassified examples. Once the weighted sum has been obtained, it is used to reduce the probability of those examples that have been correctly classified by C_i: each such probability is multiplied by the term, $\beta_i = \epsilon_i/(1 - \epsilon_i)$.

Normalizing the Probabilities Whenever the probabilities change, they have to be normalized so that their values again sum to 1 as required by the theory of probability. The easiest way to do so is by dividing each probability by the sum of all probabilities. For instance, suppose that the following probabilities have been obtained:

$$p_1 = 0.4, \quad p_2 = 0.2, \quad p_3 = 0.2$$

The sum of these values is $0.4 + 0.2 + 0.2 = 0.8$. Dividing each of the three probabilities by 0.8 will give the following normalized values:

$$p_1 = \frac{0.4}{0.8} = 0.5, \quad p_2 = \frac{0.2}{0.8} = 0.25, \quad p_3 = \frac{0.2}{0.8} = 0.25$$

It is easy to verify that the new probabilities now sum to 1:

$$p_1 + p_2 + p_3 = 0.5 + 0.25 + 0.25 = 1.0$$

Numeric Example Table 9.4 shows how *Adaboost* modifies the probabilities of the individual examples. At the beginning, each example has the same chance of being selected for the first training set. Once the first classifier has been induced, it is applied to each example from the original set, T.

The probabilities of those examples that were correctly classified by C_1 (examples $\mathbf{x}_1$ through $\mathbf{x}_7$) are reduced according to $p_{i+1}(\mathbf{x}_j) = p_i(\mathbf{x}_j) \cdot \beta_i$, where $\beta_i = \epsilon_i/(1-\epsilon_i)$ and $\epsilon_i = \sum_{j=1}^{m} p_i(\mathbf{x}_j)e_i(\mathbf{x}_j)$. The resulting probabilities are then normalized to make sure they sum to 1.

Next, the second training set, T_2, is created and classifier C_2 induced from it. Based on the observed behavior of C_2 on the entire T, we calculate the values of ϵ_2 and β_2; these are then used for the modification of the probabilities of the correctly classified examples, $p(\mathbf{x_1}) \dots p(\mathbf{x_{10}})$. The process continues until a termination criterion is reached, for instance, a predefined number of classifiers having been reached, or the classification accuracy of the whole "assembly" having achieved a certain threshold.

Table 9.4 Example of *Adaboost* modifying the probabilities of examples

Suppose the training set, T, consists of ten examples, $\mathbf{x}_1 \dots \mathbf{x}_{10}$. The number of examples being $m = 10$, all initial probabilities are set to $p_1(x_i) = 1/m = 0.1$:

$p_1(\mathbf{x}_1)$	$p_1(\mathbf{x}_2)$	$p_1(\mathbf{x}_3)$	$p_1(\mathbf{x}_4)$	$p_1(\mathbf{x}_5)$	$p_1(\mathbf{x}_6)$	$p_1(\mathbf{x}_7)$	$p_1(\mathbf{x}_8)$	$p_1(\mathbf{x}_9)$	$p_1(\mathbf{x}_{10})$
0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

According to this probability distribution, the examples for inclusion in T_1 are selected. From T_1, classifier C_1 is induced.

Suppose that, when applied to T, classifier C_1 classified correctly examples $\mathbf{x}_1 \dots \mathbf{x}_7$ (for these, $e_1(\mathbf{x}_j) = 0$) and misclassified examples $\mathbf{x}_8 \dots \mathbf{x}_{10}$ (for these, $e_1(\mathbf{x}_j) = 1$).

The weighted error is then obtained as follows:

$$\epsilon_1 = \Sigma_{j=1}^{10} \quad p_1(\mathbf{x}_j) \cdot e_1(\mathbf{x}_j) = 0.3$$

From here, the multiplicative term is calculated:

$$\beta_1 = \epsilon_1/(1-\epsilon_1) = 0.43$$

The probabilities are modified by $p(\mathbf{x_j}) = p(\mathbf{x_j}) \cdot \beta_1$

Here are the new (not yet normalized) values:

$p_2(\mathbf{x}_1)$	$p_2(\mathbf{x}_2)$	$p_2(\mathbf{x}_3)$	$p_2(\mathbf{x}_4)$	$p_2(\mathbf{x}_5)$	$p_2(\mathbf{x}_6)$	$p_2(\mathbf{x}_7)$	$p_2(\mathbf{x}_8)$	$p_2(\mathbf{x}_9)$	$p_2(\mathbf{x}_{10})$
0.043	0.043	0.043	0.043	0.043	0.043	0.043	0.1	0.1	0.1

Normalization results in the following values:

$p_2(\mathbf{x}_1)$	$p_2(\mathbf{x}_2)$	$p_2(\mathbf{x}_3)$	$p_2(\mathbf{x}_4)$	$p_2(\mathbf{x}_5)$	$p_2(\mathbf{x}_6)$	$p_2(\mathbf{x}_7)$	$p_2(\mathbf{x}_8)$	$p_2(\mathbf{x}_9)$	$p_2(\mathbf{x}_{10})$
0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.17	0.17	0.17

Note that these ten probabilities sum up to 1.0. The next classifier, C_2, is then induced from a training set T_2 whose examples have been selected from T according to these last probabilities.

Weighted Majority Voting Once the classifiers have been induced, the example to be classified is presented to them all in parallel, and the final classification decision is reached by *weighted majority voting* whereby each classifier is given a weight according to its classification record (see below). The voting mechanism in *Adaboost* is less than democratic in that it weighs the individual classifiers' strengths with coefficients denoted by w_i.

When presented with an example, each classifier returns a class label. The final decision is obtained by comparing the sum, $W_{\texttt{pos}}$, of the weights of the classifiers voting for the positive class with the sum, $W_{\texttt{neg}}$, of the weights of the classifiers voting for the negative class.

For instance, suppose there are seven classifiers with the following weights: $[0.2, 0.1, 0.3, 0.7, 0.2, 0.9, 0.8]$. Suppose, further, that the first four return `pos`, and the last three return `neg`. Plain voting would return the `pos` label because this class is supported by the greater number of votes. By contrast, *weighted majority voting* will separately sum the weights supporting the positive class, obtaining $W_{\texttt{pos}} = 0.2 + 0.1 + 0.3 + 0.7 = 1.3$ and then sum the weights supporting the negative class: $W_{\texttt{neg}} = 0.2 + 0.9 + 0.8 = 1.9$. The master classifier labels the example as negative because $W_{\texttt{neg}} > W_{\texttt{pos}}$.

Defining the Weights of Individual Classifiers Each classifier is assigned a weight according to its performance: the higher the classifier's reliability, the higher its weight. The weight can in principle even be negative—if it is believed that the classifier more often fails than succeeds. It is important to know how to find the concrete weights.

Many possibilities exist. The inventors of *Adaboost* suggested formulas that facilitated their mathematical analysis of the technique's behavior. Practically speaking, though, one can just as well use the perceptron-learning algorithm from Chap. 4. The idea is to begin with equal weights for all classifiers and then present the system, one by one, with the training examples. Each time the master classifier makes an error, we increase or decrease the weights of the individual classifiers according to the relation between the master classifier's hypothesis, $h(\mathbf{x})$, and the training example's class, $c(\mathbf{x})$.

One can also use WINNOW because it is good at weeding out classifiers that do not contribute much to the overall system's performance (as if these classifiers were irrelevant attributes describing an example).

9.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Describe the mechanism that *Adaboost* uses when selecting the training examples to be included in the training set, T_i (from which the i-th classifier is to be induced).

- Explain how, after the induction of the i-th classifier, *Adaboost* modifies for each example its probability of being chosen for inclusion in T_{i+1}.
- Explain the principle of the *weighted majority voting* that *Adaboost* uses when deciding about a concrete example's class label.
- How are the weights of the individual classifiers obtained?

9.4 Variations on the Boosting Theme

Boosting combines several imperfect classifiers that tend to complement one another. Schapire was the first to suggest a concrete way of inducing such classifiers, *bagging* and *Adaboost* being the most popular alternatives. The variations on this theme are virtually inexhaustible. Let us mention some of the most important ones.

Randomizing Attributes Rather than inducing each classifier from a different training subset, one may use the same training examples, but each time described by a different subset of attributes.

The input is the set, T, of the training examples and the set, A, of the attributes that describe them. Instead of the random subsets of examples (as in *bagging*), we choose N random subsets of attributes, $A_i \subset A$. The i-th classifier ($i \in [1, N]$) is induced from all examples from T, described by attributes from A_i. As before, the classifiers participate in *weighted majority voting*, the weights being obtained, say, by perceptron learning.

The technique is useful in domains with many attributes of which most are suspected of being either irrelevant or redundant. Classifiers induced from examples described by imperfect attributes will exhibit poor classification performance and thus receive low (or even negative) weights.

Random Forests Suppose that, for the baseline induction techniques, we choose induction of decision trees. A popular approach known as *random forests* combines *bagging* with *randomized attributes*. Not only is each of the voting classifiers induced from a different set of examples but also describes them by a different set of attributes. The names come from the fact that many decision trees are voting.

Non-homogeneous Boosting The boosting techniques presented so far have all assumed that the individual classifiers are induced from somewhat different data, but always with the same baseline learning technique. Generally speaking, however, this does not have to be the case. *Non-homogeneous boosting* does the exact opposite: each classifier is induced from the same data, but with a different machine-learning technique. The classifiers then, again, participate in *weighted majority voting*.

The main advantage of this approach is the way it reduces error. Chapter 11 will explain that the errors committed by any classifier fall into two categories. The first is due to the *variance* in the data: a different training set results in a different classifier that errs on different examples. The second source is the *bias* inherent in the classifier; for instance, a linear classifier will misclassify some examples if the two classes are not linearly separable. *Non-homogeneous boosting* is known to

Table 9.5 The class labels suggested by the six base-level classifiers are used as attributes to redescribe the examples. Each column represents a training example to be used for the induction of the master classifier

	$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	...	$\mathbf{x_m}$
Classifier 1	1	1	0	...	0
Classifier 2	0	0	1	...	1
Classifier 3	1	1	0	...	1
Classifier 4	1	1	0	...	1
Classifier 5	0	1	0	...	1
Classifier 6	0	0	0	...	1
Real class	1	1	0	...	1

reduce both kinds of error: *variance*-related errors, which are reduced in *all* boosting algorithms, and *bias*-related errors—an advantage specific to *non-homogeneous boosting*.

Stacking The *non-homogeneous boosting* from the previous paragraph takes the outputs of the individual classifiers and then reaches the final decision by *weighted majority voting*. Two layers are involved: at the lower are the base-level classifiers, and at the upper is a master classifier that combines their outputs. Note that the master classifier itself has to be induced from data, perhaps by perceptron learning because the weighted voting essentially represents a linear classifier.

In the *stacking* approach, the principle is generalized. While the lower layer, as before, employs a set of diverse classifiers, the master classifier can essentially use *any* machine-learning paradigm: Bayesian, nearest-neighbor based, a decision tree, or a neural network. The linear classifier from *non-homogeneous boosting* is just one out of many possibilities.

The base-level classifiers may each rely on a different paradigm. Very often, however, they differ only in their choice of parameter settings. For instance, there can be a few decision trees, each with a different extent of pruning, and to these may be added a few k-NN classifiers, each with a different k. What matters is that each such classifier should differ in its classification behavior.

The method is illustrated in Table 9.5. The rows represent the base classifiers: six of them were induced, each by a different technique. The columns represent the training examples. The field in the i-th row and j-th column contains 1 if the i-th classifier labels the j-th example as positive; otherwise, it contains 0. Each column can be interpreted as a binary vector that redescribes the given example by the labels assigned to it by these classifiers. This new training set is then presented to another machine-learning program that induces the master classifier.

9.4.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of *randomized attribute sets*. What is its main advantage? How can it be combined with classical *bagging*?

- What is the essence of *random forests*?
- Explain the principle of *non-homogeneous boosting*. What is its main advantage from the perspective of the errors committed by the resulting classifier?
- Explain the two-layered principle of "stacking." In what sense can stacking be seen as a generalization of *non-homogeneous boosting*?

9.5 Cost-Saving Benefits of *Boosting*

In some machine-learning algorithms, computational costs grow very fast with the growing size of the training set. An experimenter may observe that induction from half of the examples takes only a small fraction of the time that would be needed if all training examples were used. Likewise, the induction technique's costs may grow very fast with the growing number of attributes.

Illustration The situation is visualized in Fig. 9.4. Here, the time needed by a hypothetical machine-learning technique to induce a classifier from N examples is T. However, the time needed to induce a classifier from just half of the examples is only one-fifth of the time, $0.2T$. It follows that to induce two classifiers, one from the first half of the training set and the other from the second half, we will need only $2 \times 0.2T = 0.4T$ of the time that would have been needed for the induction of the classifier from the whole set, T. The computational savings thus amount to 60% of the original costs.

Generalization to K Classifiers Following the same logic, we may consider induction of K classifiers, each from a different subset, T_i, and each consisting of m examples such that m is much smaller than the size of the whole training set. The classifiers induced from these subsets will then vote, just as they do in *bagging*. In

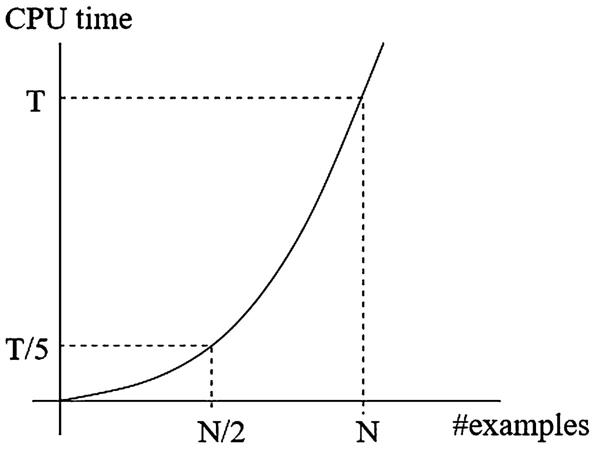


Fig. 9.4 In some techniques, computational costs grow quickly with the growing size of the training set. Quite often, induction from half of the examples incurs only a small fraction of the costs incurred by induction from all examples

many cases, the induction of the individual classifiers will take only a tiny fraction of the original time—and yet the classification performance of the whole group may compare favorably with that of a classifier induced, at great computation costs, from the entire training set. Similar observations can be made also in the case of *Schapire's boosting* and *Adaboost*.

We see that boosting not only improves classification performance but may also leads to savings in computational costs. These savings may be critical in domains marked by a great many training examples described by many attributes. Induction can then be very expensive, and any idea that helps reduce the costs is welcome.

Comments on Practical Implementation In *bagging*, the number of examples to be included in T_i is the same as in the original training set. No computational savings in the sense of Fig. 9.4 will thus be observed. However, we must not forget that the assumption about the sizes of T_i was needed only to facilitate the presentation of *bagging* as a bootstrapping technique. In practical applications, this is unnecessary: the size, m, of the sets T_i can be just another user-set constant.

The reader will recall that *Schapire's Boosting*, *non-homogeneous boosting*, and *stacking* assumed that the sizes of T_i, were chosen by the user.

Costs of Example Selection When considering these cost savings, we must not forget that we pay for them with extra overhead. More specifically, additional computational time is needed to select the examples to be included in the next T_i.

In the case of *bagging*, these costs are so small as to be easily neglected. They are higher in *Adaboost*, but even here they are usually affordable. The situation is different in *Schapire's boosting*. Here, the search for examples that satisfy the conditions for inclusion in T_2 and T_3 can be quite expensive, especially when a great number of training examples are available from which only a small percentage of them satisfy the conditions.

9.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Elaborate on the statement that "computational costs of induction can be greatly reduced by exploiting the idea of the voting assemblies."
- Discuss the circumstances under which one can expect *boosting* to reduce computational costs.
- How expensive (computationally) is it to create the training subsets, T_i, in each of the boosting techniques?

9.6 Summary and Historical Remarks

- A popular machine-learning approach induces a set of classifiers, each from a different subset of the training examples. When presented with an example, each classifier offers a class, the final decision being delegated to a master classifier.
- The simplest application of this idea is *bagging*. Here, the subsets used for the induction of the individual classifiers are obtained from the original training set, T, by bootstrapping. Suppose that T contains m examples. Then, when creating T_i, we choose m examples "with replacement." Some examples may then appear in T_i more than once, while others will not appear there at all.
- *Schapire's boosting* induces three classifiers, C_1, C_2, and C_3, making sure that they complement one another as much as possible. This complementarity is the result of the mechanism that creates the training subsets: the composition of T_1 is random, the composition of T_2 is such that C_1 experiences 50% error rate on this set, and T_3 consists of examples on which C_1 and C_2 disagree. Each of the three subsets has the same number of examples.
- *Adaboost* chooses the training subsets probabilistically in a way that makes each T_i consist primarily of examples on which the previous classifiers failed. Another difference is that the final class label is obtained by *weighted majority voting*.
- Other variations exist. One of them, *randomizing attributes*, induces the classifiers always from the same training examples which, however, are each time described by a different subset of the attributes. Another, *non-homogeneous boosting*, uses always the same training set and the same attributes but induces each classifier by a different induction technique. Both of these techniques decide about the final class by *weighted majority voting*. Finally, *stacking* resembles non-homogeneous boosting, but the output is decided by a master classifier that is more general than just linear classifier. This master classifier is trained on the outputs of the base-level classifiers.

Historical Remarks The idea of boosting was invented by Schapire (1990) who pointed out that, in this way, even the performance of very weak induction paradigms can be "boosted"—hence the name. The more practical idea underlying *Adaboost* was published by Freund and Schapire (1996), whereas the *bagging* approach was explored by Breiman (1996); its application to decision trees, known as *random forests*, was published by Breiman (2001). The principle of *stacking* (under the name *Stacking Generalization*) was introduced by Wolpert (1992).

9.7 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

Table 9.6 The probabilities of ten training examples

$p(\mathbf{x}_1)$	$p(\mathbf{x}_2)$	$p(\mathbf{x}_3)$	$p(\mathbf{x}_4)$	$p(\mathbf{x}_5)$	$p(\mathbf{x}_6)$	$p(\mathbf{x}_7)$	$p(\mathbf{x}_8)$	$p(\mathbf{x}_9)$	$p(\mathbf{x}_{10})$
0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.17	0.17	0.17

9.7.1 Exercises

1. Suppose the probabilities of the training examples to be used by *Adaboost* are those listed in Table 9.6. From these, a training subset, T_i, has been created, and from T_i, classifier C_i is induced. Suppose that C_i then misclassifies examples $\mathbf{x}_2$ and $\mathbf{x}_9$. Show how the probabilities of all training examples are recalculated and then normalize these probabilities.
2. Suppose that eight classifiers have labeled an example. Let the weights of the classifiers returning the `pos` label be $[0.1, 0.8, 0.2]$, and let the weights of the classifiers returning the `neg` label be $[-0.1, 0.3, 0.3, 0.4, 0.9]$. What label is going to be returned by a master classifier that uses *weighted majority voting*?
3. Return to Table 9.5 that summarizes the class labels returned for some of the m examples by six different classifiers. Suppose a 3-NN-based master classifier is asked to label an example and that the three nearest neighbors (the columns in Table 9.5) are $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$. What final label is returned?

9.7.2 Give it Some Thought

1. Recall how *Schapire's boosting* chooses the examples for inclusion in the training sets T_2, and T_3. Discuss possible situations under which it is impossible to find enough examples for these subsets. When will it happen that enough examples *can* be found, but the search for them is impractically expensive? Conversely, under what circumstances will it be affordable to identify all the necessary examples even when recursion is used?
2. Give some thought to the *stacking* approach. Think of a situation under which it is bound to disappoint. Conversely, suggest a situation where *stacking* will outperform the less general *non-homogeneous boosting*.

9.7.3 Computer Assignments

1. Implement the basic algorithm of *Adaboost*. The number of voting classifiers is determined by a user-set constant. Another user-set constant specifies the number of examples in each training set, T_i. The weights of the individual classifiers are obtained with *perceptron learning*.

2. Apply the program implemented in the previous task to some of the benchmark domains from the UCI repository.[2] Study this program's performance on different data. For each domain, plot a graph showing how the overall accuracy of the resulting classifier depends on the number of voting subclassifiers. Observe how the error rate on the training set and the error rate on the testing set tend to converge with the growing number of classifiers.
3. Implement the *stacking* algorithm for different base-level learning algorithms and for different types of the master classifier. Apply the implemented program to a few benchmark domains, and observe its behavior.

[2] www.ics.uci.edu/~mlearn/MLRepository.html.

Chapter 10
Classifiers in the Form of Rule-Sets

Some classifiers are best expressed in the form of *if-then* rules: if the conditions in the *if*-part are satisfied, the example is labeled with the class specified by the *then*-part. The advantage is that the rule captures the logic underlying the given class, and thus facilitates an explanation of why an example is to be labeled in this or that concrete way. Typically, a classifier of this kind is represented not by a single rule, but by a set of rules, a *rule-set*. Induction of rule-sets is capable of discovering recursive definitions, something that other machine-learning paradigms cannot do.

When discussing the techniques that induce rules or rule-sets from training data, we will rely on ideas borrowed from *Inductive Logic Programming*, a discipline that studies methods for automated development and improvement of *Prolog* programs. Here, however, we will limit ourselves only to classifier induction.

10.1 Class Described by Rules

To prepare the ground for simple rule-induction algorithms, let us take a look at the nature of the rules that interest us. After this, we will introduce the relevant terminology and define the specific machine-learning task.

Essence of Rules Table 10.1 contains the training set of the "pies" domain from Chap. 1. The following expression is one possible description of the positive class:

```
[ (shape=circle) AND (filling-shade=dark) ] OR
[ NOT(shape=circle) AND (crust-shade=dark) ]
```

When classifying example, **x**, the classifier compares the example's attribute values with those in the expression. Thus if **x** is `circular` and its `filling-shade` is `dark`, the expression is *true*, and the classifier labels **x** as positive. If the

© Springer Nature Switzerland AG 2021
M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_10

Table 10.1 Twelve training examples from the "pies" domain

		Crust		Filling		
Example	Shape	Size	Shade	Size	Shade	Class
Ex1	Circle	Thick	Gray	Thick	Dark	pos
Ex2	Circle	Thick	White	Thick	Dark	pos
Ex3	Triangle	Thick	Dark	Thick	Gray	pos
Ex4	Circle	Thin	White	Thin	Dark	pos
Ex5	Square	Thick	Dark	Thin	White	pos
Ex6	Circle	Thick	White	Thin	Dark	pos
Ex7	Circle	Thick	Gray	Thick	White	neg
Ex8	Square	Thick	White	Thick	Gray	neg
Ex9	Triangle	Thin	Gray	Thin	Dark	neg
Ex10	Circle	Thick	Dark	Thick	White	neg
Ex11	Square	Thick	White	Thick	Dark	neg
Ex12	Triangle	Thick	White	Thick	Gray	neg

expression is *false*, the classifier labels the example as negative. Importantly, the expression can be converted into the following two *rules*:

R1: *if* `[ (shape=circle) AND (filling-shade=dark) ]` *then* **pos**.
R2: *if* `[ NOT(shape=circle) AND (crust-shade=dark) ]` *then* **pos**.
else **neg**.

In the terminology of machine learning, each rule consists of an *antecedent* (the *if*-part), which in this context is a conjunction of attribute values, and a *consequent* (the *then*-part), which points to the concrete class label.

Note that the *then*-part of both rules indicates the positive class. An example is labeled as positive only if the antecedent of at least one rule is satisfied; otherwise, the classifier labels the example with the default class which, in this case, is **neg**. We will remember that when working with rule-sets in domains of this kind, we must not forget to specify the default class.

Simplifying Assumptions Throughout this chapter, we will rely on the following simplifying assumptions:

1. All training examples are described by discrete-valued attributes.
2. The training set is noise-free.
3. The training set is consistent: examples described by the same attribute vectors must belong to the same class.

Machine-Learning Task Our goal is an algorithm for the induction of rule-sets from data that satisfy the above simplifying assumptions. We will focus on rules whose consequents point to the positive class, the default always being the negative class.

Since the training set is supposed to be consistent and noise-free, we will be interested in classifiers that correctly classify all training examples. This means that for each positive example, the antecedent of at least one rule has to be *true*. For any negative example, no rule's antecedent is *true*, which is why the example is labeled with the default (negative) class.

Rule "covers" an Example Let us introduce the notion of an example being *covered* by a rule. To see the point, consider the following rule:

R: *if* `(shape=circle)` *then* **pos**.

If we apply this rule to the examples in Table 10.1, we will observe that the antecedent's condition, `shape=circle`, is satisfied by the following set of examples: $\{\mathbf{ex}_1, \mathbf{ex}_2, \mathbf{ex}_4, \mathbf{ex}_6, \mathbf{ex}_7, \mathbf{ex}_0\}$. We say that **R** *covers* these six examples. Generally speaking, a rule covers an example if the expression in the rule's antecedent is *true* for this example. Note that four of the examples covered by this particular rule are positive and two are negative.

Rule Specialization Suppose we modify the above rule by adding to its antecedent another condition, `filling-shade=dark`, obtaining the following:

R1: *if* `(shape=circle) AND (filling-shade=dark)` *then* **pos**

Checking **R1** against the training set, we realize that it covers the following examples: $\{\mathbf{ex}_1, \mathbf{ex}_2, \mathbf{ex}_4, \mathbf{ex}_6\}$. We see that this is a subset of the six examples originally covered by **R**. Conveniently, only positive (and no negative) examples are now covered.

This leads us to the definition of another useful term. If a modification of a rule's antecedent reduces the set of covered examples, we say that the modification has *specialized* the rule. In other words, specialization narrows the set of covered examples to a proper subset. A typical way of specializing a rule is by adding a new condition to the rule's antecedent.

Rule Generalization Conversely, a rule is *generalized* if its modification enlarges the set of covered examples to a superset—if the new version covers all examples that were covered by the previous version, plus some additional ones. The easiest way to generalize a rule is by removing a condition from its antecedent. For instance, this happens when we drop from **R1** the condition `(filling-shade=dark)`.

Specialization and Generalization of Rule-Sets We are interested in induction of rule-sets that label an example with the positive class if the antecedent of at least one rule is *true* for the example. This is the case of the rule-set consisting of the rules **R1** and **R2** above.

If we remove one rule from a rule-set, the rule-set may no longer cover some of the previously covered examples. This, we already know, means specialization. Conversely, adding a new rule to the rule-set will generalize the rule-set because the new rule will add to the set of covered examples.

10.1.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Explain the nature of rule-based classifiers. What do we mean when we say that a rule *covers* an example? Using this term (*cover*), specify how the induced classifier should behave on a consistent and noise-free training set.
- Define the terms *generalization* and *specialization*. How will you specialize or generalize a rule? How will you specialize or generalize a rule-set?
- List the simplifying assumptions we said we would use throughout this chapter.

10.2 Inducing Rule-Sets by Sequential Covering

Let us introduce a simple technique to induce rule-sets from training data that satisfy the simplifying assumptions from the previous section.

Principle The goal is to find a rule-set such that each of its rules covers some positive examples, but no negative examples. Together, the rules should cover all positive examples and no negative ones. The procedure creates one rule at a time, always starting with a very general initial version (which covers also negative examples) that is then gradually specialized until all negative examples are excluded. The circumstance that the rules are created sequentially and that each is supposed to cover those positive examples that were missed by previous rules gives the technique its name: *sequential covering*.

Baseline Version of Sequential Covering Table 10.2 contains the pseudo-code of a simple technique to induce the rule-sets. Sequential covering constitutes the main

Table 10.2 Pseudo-code of *sequential covering*

```
Input: training set $T$.
Sequential covering.
Create an empty rule-set.
While at least one positive example remains in $T$,
  1. Create a rule using the algorithm below.
  2. Remove from $T$ all examples that satisfy the rule's antecedent.
  3. Add the rule to the rule-set.

Create a single rule
Create an initial "empty" version of the rule, R: if (), then pos
  1. If R does not cover any negative example, stop.
  2. Add to R's antecedent a condition, $a_i = v_j$, and return to the previous step.
```

part. Each rule covers some positive examples, but no negative examples. Once the rule has been created, the examples covered by it are removed from the training set. If no positive examples remain, the algorithm stops; otherwise, the algorithm is applied to the reduced training set.

The lower part of the table describes induction of a single rule. The algorithm starts with the most general version of the antecedent: it says, "all examples are positive." Assuming that the training set contains at least one negative example, this statement is obviously incorrect. The algorithm seeks to rectify the situation by *specialization*, trying to exclude from coverage some negative examples, hopefully without losing the coverage of the positive examples. The specialization operator adds to the rule another conjunct in the form, $a_i = v_j$ (read: the value of attribute a_i is v_j).

Illustration Let us hand-simulate the sequential-covering algorithm using the data from Table 10.1. The first rule, the one with the empty antecedent, covers all training examples. Adding to the empty antecedent, the condition `shape=circle` results in a rule that covers four positive and two negative examples. Adding one more condition, `filling-shade=dark`, specializes the rule so that, while still covering the four positive examples, it no longer covers any negative example. We have obtained a rule that covers examples $\{\mathbf{ex}_1, \mathbf{ex}_2, \mathbf{ex}_4, \mathbf{ex}_6\}$. This is the rule **R1** from the previous section.

If we remove these four examples from the training set, we are left with only two positive examples, $\mathbf{ex}_3$ and $\mathbf{ex}_5$. The development of another rule again starts from the most general version (empty antecedent). Suppose that we then choose `shape=triangle` as the initial condition. This covers one positive and two negative examples. Adding to the antecedent the term `filling-shade=dark` excludes the negative examples while retaining the coverage of the positive example $\texttt{ex}_3$, which can now be removed from the training set. Once this second rule has been created, we are left with one positive example $\texttt{ex}_5$.

We therefore have to create yet another rule whose task will be to cover $\texttt{ex}_5$ without covering any negative example. Once we find such rule, $\texttt{ex}_5$ is removed from the training set. Since there are no positive examples left, the procedure is terminated. We have created a rule-set consisting of three rules that cover all positive examples and no negative example.

How to Identify the Best Attribute-Value Pair In our example, when specializing a rule, we chose the condition to be added to the rule's antecedent more or less at random. But since we usually can select the added condition from a set of alternatives, we need a mechanism to inform us about the quality of each choice. One natural criterion is based on information theory, a principle we already know from Chap. 5 where it was used to select the most informational attribute.

Let N_{old}^{+} be the number of positive examples covered by the rule's original version, and let N_{old}^{-} be the number of negative examples covered by the rule's original version. Likewise, the numbers of positive and negative examples covered by the rule's new version will be denoted by N_{new}^{+} and N_{new}^{-}, respectively.

Since the rule covers only positive examples, the information content of the message that the rule labels a randomly picked example as positive is calculated as follows (for the old version and for the new version):

$$I_{old} = -\log(\frac{N_{old}^+}{N_{old}^+ + N_{old}^-})$$

$$I_{new} = -\log(\frac{N_{new}^+}{N_{new}^+ + N_{new}^-})$$

The difference between the two quantifies the information gained by the rule modification. Machine-learning professionals usually weigh the information gain by the number of covered examples, N_C, so as to give preference to rule modifications that optimize the number of covered examples. The quality of the rule improvement is then calculated as follows:

$$Q = N_C \times |I_{new} - I_{old}| \tag{10.1}$$

When comparing alternative ways of modifying a rule, we choose the one with the highest value of Q.

10.2.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Summarize the principle of the *sequential covering* algorithm.
- Explain the mechanism of gradual rule specialization. What do we want to accomplish by this specialization?
- How will you exploit information gain when looking for the most promising way of specializing a rule?

10.3 Predicates and Recursion

The *sequential covering* algorithm has a much broader scope of applications than the previous section indicated. Importantly, the technique is capable of inducing concepts expressed in predicate calculus.

Predicates: Greater Expressive Power Than Attributes A serious limitation of attribute-value logic is that it is not flexible enough to capture certain relations among data. For instance, the fact that **y** is located between **x** and **z** can be stated by the predicate `between(x,y,z)`.[1] To express the same relation by means of attributes and their values would be difficult.

An attribute is a special case of a one-argument predicate. For instance, the fact that, for a given example, `x`, the shape is `circular` can be written as `circular(x)`. But the analogy is no longer obvious when it comes to predicates with two or more arguments.

Induction of Rules in Predicate Calculus Here is an example of a rule that says that if `x` is a parent of `y`, and at the same time `x` is a woman, then this parent is `y`'s mother:

if `parent(x,y)` AND `female(x)` *then* `mother(x,y)`

We can see that this rule has the same structure as the rules **R1** and **R2** we have seen before: a list of conditions in the antecedent is followed by a consequent. And indeed, the same sequential covering algorithm can be employed for induction. There is one difference, though. When choosing among the candidate predicates to be added to the antecedent, we must not forget that the meaning of the predicate changes if we change the arguments. For instance, the previous rule's meaning will change if we replace `parent(x,y)` with `parent(x,z)` because, in this case, the fact that `x` is a parent of `z` surely does not guarantee that `x` is mother of some other subject, `y`.

Rule-Sets Facilitate Recursive Definitions The rules can be more interesting than those in the toy domain from Table 10.1 might have led us to believe. For one thing, they can be *recursive*—as in the case of the following two rules that define an `ancestor`.

if `parent(x,y)` *then* `ancestor(x,y)`

if `parent(x,z)` AND `ancestor(z,y)` *then* `ancestor(x,y)`

The meaning is easy to see. Ancestor is a parent or at least a parent's ancestor. For instance, a grandparent is the parent of a parent—and therefore an ancestor.

Example of Induction Let us illustrate induction of rule-sets using the problem from Table 10.3. Here, two concepts, `parent` and `ancestor`, are characterized by a list of positive examples under the assumption that any example that is not in this list should be regarded as a negative example. Our goal is to induce the definition of `ancestor`, using the predicate `parent`.

We begin with the most general rule, *if* () *then* `ancestor(x,y)`. In the next step, we want to add a condition to the antecedent. We may consider various possibilities, but the simplest appears to be `parent(x,y)`—which will also be recommended by the information-gain criterion. We have obtained the following rule:

[1]More accurately, the predicate is the term "`between`," whereas `(x,y,z)` is a list of arguments.

Table 10.3 Illustration of induction from examples described in predicate logic

Consider the knowledge base consisting of the following positive examples of classes `parent` and `ancestor`, defined as Prolog-like facts (any example absent from this list is deemed negative).

`parent(eve,ted)`	`ancestor(eve,ted)`	`ancestor(eve,ivy)`
`parent(tom,ted)`	`ancestor(tom,ted)`	`ancestor(eve,ann)`
`parent(tom,liz)`	`ancestor(tom,ted)`	`ancestor(eve,jim)`
`parent(ted,ivy)`	`ancestor(tom,ted)`	`ancestor(tim,ivy)`
`parent(ted,ann)`	`ancestor(tom,ted)`	`ancestor(eve,ann)`
`parent(ann,jim)`	`ancestor(tom,ted)`	`ancestor(eve,jim)`
		`ancestor(ted,jim)`

From these examples, the algorithm creates the following first version of the rule. Note that this rule does not cover any negative examples.

R3: *if* `parent(x,y)` *then* `ancestor(x,y)`

After the removal of all positive examples covered by this rule, the following positive examples of `ancestor(x,y)` remain:

```
ancestor(eve,ivy)
ancestor(eve,ann)
ancestor(eve,jim)
ancestor(tim,ivy)
ancestor(eve,ann)
ancestor(eve,jim)
```

To cover these, another rule is created:

if `parent(x,z)` *then* `ancestor(x,y)`

After specialization, this second rule is turned into the following:

R4: *if* `parent(x,z)` AND `ancestor(z,y)` *then* `ancestor(x,y)`

The two rules **R3** and **R4** now cover all positive examples and no negative examples.

R3: *if* `parent(x,y)` *then* `ancestor(x,y)`

Observing that the rule covers only positive examples and no negative examples, we realize there is no need to specialize it.

However, the rule covers only the `ancestor` examples from the middle column and no examples from the right column, which means that we need at least one more rule. When considering the conditions to be added to the empty antecedent of the next rule, we may consider the following (note that this is the same predicate, but each time with a different set of arguments):

```
parent(x,z)
parent(z,y)
```

Suppose that the first of the two provides higher information gain. Seeing that the rule still covers some negative examples, we will specialize it by adding another condition to its antecedent. Since the `parent` predicate does not lead us anywhere, we try `ancestor`, again with diverse lists of arguments. Evaluating the information gain of all alternatives, we learn that the best option is `ancestor(z,y)`. Here is the obtained second rule:

R4: *if* `parent(x,z)` AND `ancestor(z,y)` *then* `ancestor(x,y)`.

10.3.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- How can a concept be expressed in predicate logic? In what sense are predicates richer than attributes?
- Give an example of a recursively defined concept. Can you think of something else than `ancestor`?

10.4 More Advanced Search Operators

The technique described in the previous sections followed a simple strategy: seeking to find a good rule-set, the algorithm sought to modify the rule(s) by specialization and generalization, evaluating alternative options by the information-gain criterion.

Operators for Rule-Set Modification In reality, the search can be more flexible than the one discussed in the previous section. Other rule-set-modifying operators have been suggested. These, as we will see, do not necessarily represent specialization or generalization, but if we take a look at them, we realize that they make sense. Let us mention in passing that these operators have been derived with the help of a well-known principle from logic, the so-called *inverse resolution*. For our specific needs, however, the method of their derivation is unimportant.

In the following, we will simplify the formalism by writing a comma instead of AND and using an arrow instead of the *if-then* construct. In all of the four cases, the operator converts the rule-set on the left into the rule-set on the right. The leftmost column gives the traditional names of these operators.

– identification: $\begin{Bmatrix} b, x \rightarrow a \\ b, c, d \rightarrow a \end{Bmatrix} \Rightarrow \begin{Bmatrix} b, x \rightarrow a \\ c, d \rightarrow x \end{Bmatrix}$

– absorption: $\begin{Bmatrix} c, d \rightarrow x \\ b, c, d \rightarrow a \end{Bmatrix} \Rightarrow \begin{Bmatrix} c, d \rightarrow x \\ b, x \rightarrow a \end{Bmatrix}$

– inter-construction: $\left\{\begin{array}{l} v, b, c \rightarrow a \\ w, b, c \rightarrow a \end{array}\right\} \Rightarrow \left\{\begin{array}{l} u, b, c \rightarrow a \\ v \rightarrow u \\ w \rightarrow u \end{array}\right\}$

– intra-construction: $\left\{\begin{array}{l} v, b, c \rightarrow a \\ w, b, c \rightarrow a \end{array}\right\} \Rightarrow \left\{\begin{array}{l} v, u \rightarrow a \\ w, u \rightarrow a \\ b, c \rightarrow u \end{array}\right\}$

Note that these replacements are not deductive: the rules on the right are never perfectly equivalent to those on the left. And yet, they do appear to make sense intuitively.

How to Improve Existing Rule-Sets? The operators from the previous paragraph can be used to improve rule-sets that have been induced by the sequential covering algorithm. We can even consider a situation where several different rule-sets have been induced.

These rule-sets can then be improved by the operators listed above. The evaluation function may give preference to more compact rules that classify correctly some auxiliary set of training examples meant to represent a concrete application domain.

10.4.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- List the rule-set-modifying operators from this section. Which field of logic has helped derive them?
- Suggest a way of using these operators when trying to improve a rule-set.

10.5 Summary and Historical Remarks

- Classifiers can be expressed in terms of rules. A rule consists of an antecedent (a list representing a conjunction of conditions) and a consequent (a class label). If, for the given example, the rule's antecedent is *true*, then the example is labeled with the label indicated by the consequent.
- If a rule's antecedent is *true*, for an example, we say that the rule *covers* the example.
- In rule induction, we often rely on *specialization*. This reduces the set of covered examples to a subset. A rule is specialized if we add a condition to its antecedent. Conversely, *generalization* enlarges the set of covered examples.

- Usually, we induce a set of rules, a *rule-set*. The classifier labels an example, **x**, as positive if the antecedent of at least one of the rules is *true* for **x**. Adding a rule to a rule-set results in generalization. Removing a rule from a rule-set results in specialization.
- The chapter introduced a simple algorithm for induction of rule-sets from noise-free and consistent training data described by discrete attributes. The algorithm can to some degree be optimized using a criterion from information theory.
- The same algorithm can be used for induction of rules in domains where the examples are described in *predicate* calculus. Even recursive rules can thus be discovered.
- Certain other operators have been developed by the logical mechanism called *inverse resolution*. They do not necessarily represent specialization or deduction.

Historical Remarks Induction of rules is one of the oldest tasks of machine learning; its origins can be traced back to the days when the discipline's primary ambition was to create the knowledge bases for artificial-intelligence systems. The sequential-covering algorithm is a simplified version of a technique proposed by Clark and Niblett (1989). Its use for induction of predicate-based rule was inspired by the FOIL algorithm developed by Quinlan (1990). The additional operators from Sect. 10.4 are based on those introduced by Muggleton and Buntine (1988) in the framework of their work on *inverse resolution*.

10.6 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

10.6.1 Exercises

1. Hand-simulate the algorithm of *sequential covering* for the data from Table 10.1. Ignoring information gain, indicate how the first rule is created if we start from `crust-shade=gray`.
2. Show that, when we choose different ways of specializing a rule (adding different attribute-value pairs), we obtain a different rule-set that may even have a different size.

10.6.2 Give it Some Thought

1. Think of some other concepts (different from those discussed in this chapter) that will with advantage be defined recursively.
2. Give some thought to the fact that some concepts are naturally recursive. Try to figure out whether they could be addressed by attribute-value logic. In this way, demonstrate the superior power of the predicate calculus.
3. Suggest a learning procedure for "knowledge refinement." In this task, we assume that certain concepts have already been defined in predicate calculus. When presented with another set of examples, the knowledge-refinement technique seeks to optimize the existing rules, either by making them more compact or by making them more robust in the presence of noise.

10.6.3 Computer Assignments

1. Write a computer program that implements the *sequential covering* algorithm. Use some simple criterion (not necessarily information gain) to choose which condition to add to a rule's antecedent.
2. In the UCI repository, find a domain satisfying the criteria specified in Sect. 10.1. Apply to it the program developed in the previous step.
3. How would you represent two-argument or three-argument predicates if you wanted to implement your machine-learning program in *C++*, *Java*, or some other general-purpose language?
4. Write a program that applies the *sequential covering* algorithm to examples described in predicate calculus.

Chapter 11
Practical Issues to Know About

To facilitate the presentation of machine-learning techniques, this book has so far neglected certain practical issues that are non-essential for beginners, but cannot be neglected in realistic applications. Now that the elementary principles have been explained, time has come to venture beyond the basics.

The first thing to consider is the *bias* that helps reduce the number of possible solutions. Next comes the observation that a larger training set may hurt the learner's chances if most of the examples belong to just one class. Another important question is how to deal with classes whose meanings vary with context or time. Finally, the chapter will address some more mundane aspects such as unknown attribute values, the need to select useful attributes or create higher-level features, as well as some other issues.

11.1 Learner's Bias

Chapter 7 concluded that "there is no learning without bias." The reasons were mathematical: an unconstrained (and hence very large) hypothesis space is bound to contain hypotheses that by mere chance may correctly classify the entire training set while erring miserably on future examples. Here is a practical view: to be able to find something, you need to know where to look; and the smaller the place where that something is hidden, the higher the chances of succeeding.

Illustration Suppose we are to identify the property shared by the following set of integers: {2, 3, 10, 12, 20, 21, 22, 28}. These are positive examples, besides which we are also provided with some negative examples where the property is absent: {1, 4, 5, 11}.

When trying to solve this problem, most people will start by various ideas related to numbers, such as primes, odd numbers, integers exceeding certain thresholds, results of arithmetic operations, and so on. With moderate effort, some

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_11

property satisfying the training examples is found. Usually, though, this solution is complicated, awkward—and unconvincing.

Here is the correct answer: all positive examples begin with `t`: two, three, . . ., all the way up to twenty-eight. Negative examples begin with different letters.

The reason most people fail to solve this puzzle is that they are looking in the wrong place. Put in the language of machine learning, they choose a wrong *bias*. Once they are shown the correct answer, their mindset will take it into account, and if they are later presented with a similar task, they will subconsciously think not only about arithmetic operations but also about language. In other words, they will incorporate this new *bias* into their thinking.

Representational Bias Versus Procedural Bias As far as machine learning is concerned, biases come in different forms. A *representational bias* is determined by the language in which we want the classifier to be formulated. Thus in domains with continuous-valued attributes, one possible representational bias is the choice of a linear classifier, another the preference for polynomials, and yet another the preference for neural networks. If all attributes are discrete-valued, the engineer may prefer conjunctions of attribute values, or perhaps decision trees. All of these biases have their advantages as well as shortcomings.

Apart from this, there is also a *procedural bias*: preference for a certain method of solution. For instance, one such bias relies on the assumption that pruning will improve the classification performance of a decision tree on future data. Another is the choice of the parameters in a neural-network training. And yet another, in the case of linear classifiers, is the engineer's decision to use *perceptron learning* rather than WINNOW.

Strength of a Bias Versus Its Correctness Suppose the engineer is to choose between a linear classifier and a neural network. If the positive and negative examples are linearly separable, then the linear classifier is better: while both paradigms contain the solution, the neural network may overfit the training set, thus poorly generalizing to future data. Conversely, if the boundary separating the two classes is highly non-linear, the linear classifier will lack flexibility, and the neural network should be chosen.

The reader has noticed two critical aspects: strength and correctness. A bias is *strong* if it implies a narrow class of classifiers. For instance, the bias of linear classifiers is stronger than that of neural networks: the former are constrained to linear decision surfaces, whereas the latter can model virtually *any* decision surface.

A bias is *correct* if it allows the solution: the linear classifier's bias is correct in domains where the positive examples are linearly separable from the negative. A conjunction of Boolean attributes is correct only if the underlying class indeed can be described by such a conjunction.

Rule of Thumb: Occam's Razor The engineer wants to use a correct bias (representational or procedural). Given the choice between two correct biases, the stronger one is preferred—a principle we know from Chap. 7 as *Occam's Razor*.

Unfortunately, we rarely know in advance which of the available biases are correct and which are incorrect; an educated guess is our only chance. In some paradigms, say, high-order polynomials, the bias is so weak that a classifier from this class has a high chance of zero error rate on the training set, and yet its performance on future data is poor. Strengthening the bias (say, by reducing a polynomial's order) will reduce the VC-dimension, thus increasing the chances on future data—but only if the bias remains correct. At a certain point, strengthening the bias further will do more harm than good because the bias becomes incorrect perhaps very much so.

We need to understand the trade-off: a mildly incorrect but strong bias can be better than a correct but weak one. However, what constitutes "a mildly incorrect bias," in a concrete application can usually be decided only by the engineer's experience or by experimentation (see Chap. 12).

Lifelong Learning We may face the necessity to learn a whole series of concepts that are all expected to require the same bias. This, for instance, was the case of the puzzle that opened this section. In applications of this kind, it makes sense to organize the learning procedure in two tiers. At the lower, the primary task is to identify the most appropriate bias, and at the higher, the software induces the classifier within the discovered bias. The term used for this strategy, *lifelong learning*, reminds us of our own human experience: the need to "learn how to learn" in a given field.

Two Sources of Error: Variance The classifier's error comes from two principal sources. The first is the *variance* in the training examples. The thing is that the data used for the induction of the classifier almost never capture all aspects of what we want to learn. In some applications, the training examples are selected at random. In others, one can use only those available at the moment, which involves a great deal of randomness, as well. In yet others, the training set has been put together by an expert whose choice was informed but subjective.

Be it as it may, one can imagine the existence of different training sets for the same domain. And here is the point. From a different training set, a somewhat different classifier is induced, and this different classifier will lead to different errors on future data. Variance in the training data is thus one important source of errors. Its negative effect is lower if we have really very large training sets.

The second source of error is *bias-related*. If the two classes, positive and negative, are not linearly separable, then any linear classifier is bound to misclassify some examples. Bias-related errors cannot be reduced below a certain limit that is determined by the nature of the given type of classifier.

Trading Bias for Variance? It is instructive to consider the *trade-off* between the two sources. The bias-related error can be reduced by choosing a machine-learning paradigm with a weaker bias; this, however, increases variance, the other source of error. Conversely, variance can be reduced by strengthening the bias, which means higher frequency of bias-related errors.

11.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the difference between the representational and procedural biases. Illustrate each type by examples.
- Explain the difference between the strong and weak biases. Explain the difference between the correct and incorrect biases. Discuss the interrelation of the two dichotomies.
- What did this section say about the two typical causes of a classifier's poor performance on future data?

11.2 Imbalanced Training Sets

When discussing the oil-spill domain, Sect. 8.3 pointed out that well-documented images of oil spills were rare. Indeed, the project could only rely on a few dozen positive examples, whereas negative examples were abundant. Such imbalanced representation of the two classes is not without serious consequences.

Simple Experiment Suppose we have at our disposal a very small training set, consisting of only 50 positive examples and 50 negative examples. Let us subject the data to fivefold cross-validation:[1] we divide the set of 100 training examples into five equally sized parts, each containing 20 examples; then, in five different experiments, we always remove one part, induce a classifier from the union of the remaining four (i.e., the union contains 80 examples), and test the classifier on the removed part. In this way, we will minimize variance-related errors. Once finished, we average the results: classification accuracy on the positive examples, classification accuracy on the negative examples, and the geometric mean of the two.

Suppose we are later provided with additional negative examples. Wishing to see how helpful they may be, we add to the previous training set these 50 negative examples (the positive examples remaining the same), repeat the experimental procedure from the previous paragraph, and then note the new results. The story then continues: we keep adding to the training set one batch of fifty negative examples after another while keeping the same original fifty positive examples.

Observation When we plot the results of these experiments in a graph, we obtain curves that look something like those in Fig. 11.1, where the 1-NN classifier was employed. The reader can see that with the growing number of the majority-class examples, the induced classifiers become biased toward this class, gradually converging to a situation where the classification accuracy on the majority class

[1] An evaluation methodology discussed in Sect. 12.5.

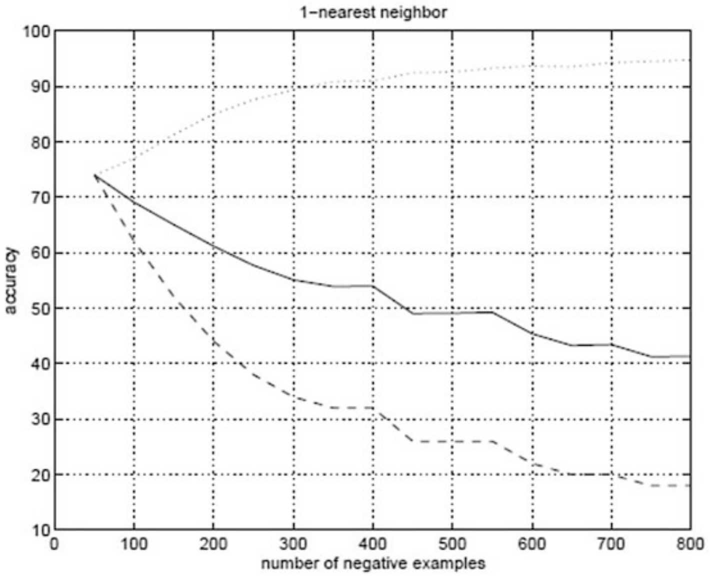


Fig. 11.1 Dotted: classification accuracy on the majority class; dashed: classification accuracy on the minority class; and solid: geometric means of the two

approaches 100%, while the classification accuracy on the minority class drops below 20%. The geometric mean of the two values keeps going down, as well.

The observation may seem counter-intuitive. Surely the induced classifier should benefit from the opportunity to learn from more examples, even if all those newly added examples are from the same class? Surprisingly, the unexpected behavior is typical of many machine-learning techniques. Experts call this the problem of *imbalanced class representation.*

Why Does It Happen in Bayesian Classifiers? To see the reason why Bayesian classifiers suffer in imbalanced domain, the reader should recall that they label example $\mathbf{x}$ with class c_i that maximizes the following product:

$$P(c_i) \cdot P(\mathbf{x}|c_i)$$

In a heavily imbalanced training set where most examples are labeled with class c_i, this class's prior probability, $P(c_i)$, is high, which means that also the above product is high. This is why most examples will be labeled with c_i.

Why Does It Happen in k-NN Classifiers? Consider a training set where the vast majority of the examples are negative, and suppose the data suffer from high-level class-label noise. In this event, class-label noise makes the nearest neighbor of almost every positive example negative (because so many positive examples have been given wrong labels when the training set was being created). Consequently, the 1-NN classifier misclassifies many positive examples. This results in numerous false negatives and few, if any, false positives. When more nearest neighbors are used, $k > 1$, the problem is mitigated, but only up to a certain degree.

What About Decision Trees and Neural Nets? In decision trees, the problem is in principle the same as in nearest-neighbor classifiers. Once the decision tree has been induced, most of the instance space tends to be covered by decision tree branches that define the majority class. As a result, the examples to be classified in the future only rarely satisfy the conditions (in the tree tests) for the minority class.

In neural networks, the situation is similar to the one experienced in Bayesian classifiers. The problem is caused by the circumstance that algorithms for MLP training seek to minimize the mean squared error (MSE). If most examples belong to one class, then MSE is minimized by a classifier that rarely, if ever, classifies an example with the minority class.

Imbalanced Classes are Common The previous paragraphs convinced us that adding to the training set only examples from the majority class may do more harm than good. This is a serious shortcoming because imbalanced training sets are common than the reader may suspect. In the oil-spill domain, the minority class was the oil spills, the primary target of the project; as already explained, these are rare. In medical diagnosis, any disease is likely to be a minority class. Also fraudulent credit card payments are relatively infrequent in the available data. In all these fields, the training set will thus be heavily imbalanced, the minority class being underrepresented.

Most of the time, it is the minority class that matters, but its examples cannot be provided in sufficient quantity. This happens either because the event in question is indeed rare (for instance, data related to kidney transplants) or because positive examples are so expensive that the user cannot afford purchasing enough of them. In the case of the oil-spill domain, both problems conspired: images of proved oil spills were rare—as well as expensive.

11.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What does the term *imbalanced training set* refer to?
- What are the causes of poor performance of classical techniques in domains with imbalanced classes?
- Provide examples of domains where the class representation are inevitably imbalanced. What can be the reasons behind rare positive examples?

11.3 Dealing with Imbalanced Classes

Many techniques for dealing with imbalanced class representation have been developed. In principle, they fall into three major categories: modification of the induction techniques, majority-class undersampling, and minority-class oversampling.

Modifying Classical Approaches In linear classifiers, the problem can to a great degree be mitigated by shifting the threshold, θ (or the bias, $w_0 = -\theta$) toward the region occupied by the majority-class examples. The same is sometimes done in the case of support vector machines (SVMs).

In nearest-neighbor approaches, one can consider modifying the classification rule. For instance, the plain 7-NN classifier labels example **x** as positive if at least 4 out of the 3 nearest neighbors of **x** in the training set are positive. If the positive class is the majority class, however, the engineer may request that **x** be labeled with the negative class even if, say, only 2 of the nearest neighbors are negative.

In the case of Bayesian classifiers and neural networks, the classifier can be instructed to issue the minority-class label even if the minority class loses—provided that it loses by a small margin. For instance, if the output of the positive-class neuron is 0.7 and the output of the negative-class neuron is 0.6, the classifier still issues the negative class.

Majority-Class Undersampling: Mechanical Approach Suppose we have at our disposal a heavily imbalanced training set where, say, nine out of ten examples are negative. In this event, the experience from Sect. 11.2 suggests that we should somehow reduce their number.

The simplest way to do so is to rely on purely random choice. For instance, each negative example may face a 50% chance of being deleted. As noticed above, the classifier induced from this reduced training set is likely to outperform a classifier induced from the original data. However, the mechanical approach will hardly satisfy an engineer who wants to understand *why* the data-removing trick worked—or, conversely, why adding more majority-class examples would hurt the classifier.

Better Solution: One-Sided Selection The nature of the trouble suggests a remedy. Seeing there are so many suspicious negative examples in the positive region, we realize we should perhaps remove primarily *these* examples (rather than resorting to the random selection of the mechanical approach).

Chapter 3 presented a technique that can identify suspicious examples: *Tomek Links*. For two examples, (**x**, **y**), to participate in a Tomek Link, three conditions have to be met: (1) each of the two examples has a different class label, (2) the nearest neighbor of **x** is **y**, and (3) the nearest neighbor of **y** is **x**. In the left part of Fig. 11.2, many of the noisy examples indeed do participate in Tomek Links. The classifier's behavior may thus improve by the deletion from the training set of the negative participants of each Tomek Link pair.

The principle is known as *one-sided* selection because only one side of the Tomek Link is allowed to remain in the training set. Applying the technique to the situation

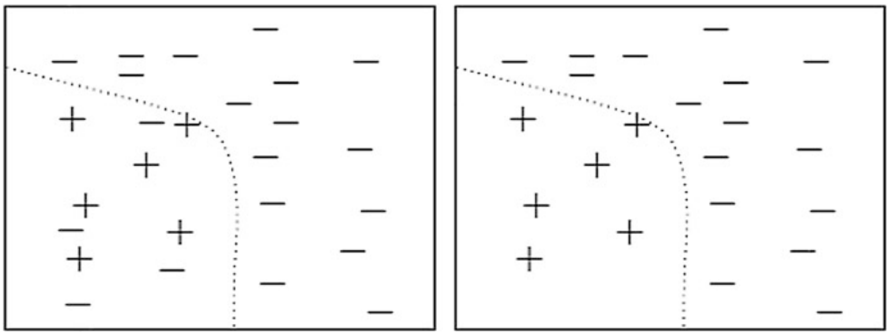


Fig. 11.2 In noisy domains where negative examples heavily outnumber positive examples, the removal of negative examples that participate in Tomek Links may improve classification performance

on the left of Fig. 11.2, we obtain the smaller training set shown on the right. The reader will agree that the false negatives are now less frequent.

One-sided selection usually outperforms the mechanical approach from above.

Minority-Class Oversampling In some domains, the training set is so small that any further reduction of its size by undersampling would be counterproductive. In a situation where even the majority-class examples are sparse, deleting any single one may remove some critical aspect of the learning task, thus negatively affecting the performance of the induced classifier.

Under these circumstances, the opposite approach is recommended. Instead of removing majority-class examples, we may prefer to *add* examples of the minority class. Not having at our disposal real examples to be added, we create them artificially. This can be done in two ways:

1. For each minority-class example, create one or more copies and add these copies to the training set.
2. For each minority-class example, create its slightly modified version and add it into the training set. The modification is caused by small random changes (noise) in continuous-valued attributes. Much less useful, though still possible, are changes in discrete attribute values.

Another look at Fig. 11.2 helps explain why this works. In the neighborhood of some positive examples whose nearest neighbors are negative only because of noise, minority-class oversampling will insert additional positive examples. As a result, the k-NN classifier is no longer misled. The approach can improve the behavior of other types of classifiers, as well.

11.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How can classical machine-learning techniques be modified to be able to deal with domains with imbalanced class representation?
- What is the essence of majority-class undersampling? Explain the mechanical approach, and then proceed to the *one-sided selection* based on Tomek Links.
- Explain the principle of minority-class oversampling. Describe and discuss the two alternative ways of creating new examples to be added to the training set.

11.4 Context-Dependent Domains

Up till now, we tacitly assumed that the underlying meaning of a given class is fixed and immutable, and that the classifier, once induced, will under all circumstances exhibit the same (or at least similar) behavior. This, however, is not always the case.

Context-Dependent Classes The meanings of some classes can be modified by changed circumstances. This is the case in our daily life, too. "Fashionable dress" depends on time and geography. "State-of-the-art technology" does not look the same today as it did a 100 years ago. Even the understanding of such commonly used terms as "democracy" or "justice" depends on political and historical background. For more technical case, consider speech-recognition systems that are known to be affected by the different pronunciations, say, in the U.K. and in the U.S.

Context-Dependent Features For our needs, *context* will be a "feature that, when considered in isolation, has no bearing on the class, while affecting the class when combined with other features."

Suppose you want to induce a classifier for medical diagnosis. Here, attribute `gender` does not mean causal relation; the patient being `male` is no proof of his suffering from `prostate-cancer`, but `gender=female` is a clear indication that the class is *not* `prostate-cancer`. This, of course, is somewhat extreme. In other diseases, the impact of `gender` will usually only affect the interpretation of certain laboratory tests, such as $p = 0.5$ being a critical threshold for male patients and $p = 0.7$ for female patients. Alternatively, prior probabilities will be modified; for Instance, `breast-cancer` is more typical of females, although it does occur in men, too. Still, the impact of context is in many domains indisputable.

Induction in Context-Dependent Domains Suppose you want to induce a speech-recognition system based on training examples both from British and from American speakers. Suppose the attribute vector describing each example contains also the context, the speaker's origin. The other attributes capture the necessary phonetic features of the concrete digital signal. Each class label represents a phoneme.

For the induction of a classifier that for each attribute vector returns the phoneme it represents, machine learning can follow two alternative strategies. The first takes advantage of the contextual attribute and divides the training examples into two subsets, one for British speakers and one for American speakers; then, it induces a separate classifier from each of these training subsets. The other strategy mixes all examples in one big training set and induces a single "universal" classifier.

Practical experience shows that, in applications of this kind, the first strategy usually performs better, assuming that the real-time system employing the contextual classifiers always knows which of them to use.

Meta-Learning In some domains, it is possible to identify for each training example its context, but this information is not available in the case of future examples that we will want to classify. The training-set information about context can then be treated as a class, and some machine-learning technique can induce a classifier capable of identifying the context. When classifying a future example, the software then first decides the example's context and then employs the classifier corresponding to this context.

This scenario is sometimes called *meta-learning.*

Concept Drift Sometimes, the context changes in time. "Fashionable dress" belongs to this category, and so do various political terms. In this event, machine-learning specialists talk about *concept drift*. What they have in mind here is that, in the course of time, the meaning of a class drifts from one context to another.

The drift has many aspects. One of them is the extent to which the meaning of the class has changed. In some rare domains, this change is so serious that the induced classifier becomes all but useless, and a new one has to be induced. Much more typical, however, is a less severe change that results only in a minor degradation of the classifier's performance. The old classifier can then still be used, perhaps after minor fine-tuning.

A critical aspect worth consideration is the "speed" of the drift. At one extreme is an abrupt change: at a given moment, one context is totally replaced by another. More typically, though, the change is gradual in the sense that there is a certain transition period during which one context is, step by step, replaced by another. In this event, the engineer may ask how fast the transition is and whether (and when) the concept drift necessitates a concrete action.

Induction in Time-Varying Domains Perhaps the simplest scenario is the one shown in Fig. 11.3. Here, a classifier is faced with a stream of examples that arrive one at a time, either in regular or in irregular intervals. Each time an example arrives, the classifier labels it with a class. A feedback loop may then tell the system (immediately or after some delay) whether the classification was correct and if not what the correct class was.

If there is a reason to suspect occasional concept drift, it may be a good idea to use a sliding window as shown in the picture. The classifier is then induced only from the examples "seen through the window." Each time a new example arrives, it is added to the window. Whenever necessary, older examples are removed, either

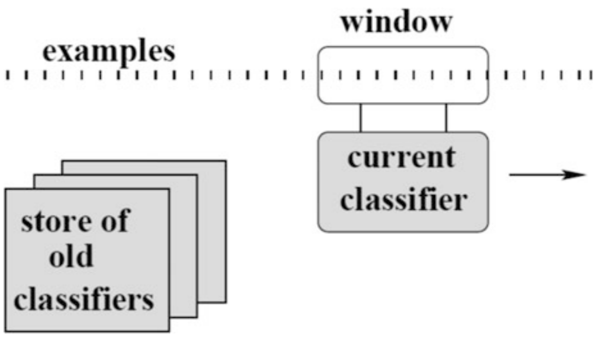


Fig. 11.3 A window passes over a stream of examples; "current classifier" is periodically updated to reflect changes in the underlying class. Occasionally, the system can retrieve some of the previous classifiers if the underlying context recurs

one at a time or in groups, such as "delete the oldest 25% examples." The motivation is simple: we need the window to contain only recent examples because older ones may be obsolete and thus unreliable.

As already mentioned, the classifier is supposed to represent only the examples in the window. In the simplest implementation, the classifier is re-induced each time the contents of the window change. Alternatively, the change in the window contents may only trigger a modification/adaptation of the classifier.

Figure 11.3 shows yet another aspect of this scenario: an older context may reappear (e.g., due to certain "seasonality"). It may then be a good idea to store previously induced versions of the classifier, just in case they may prove useful in the future.

Engineering Issues in the Sliding-Window Scenario The engineer has to consider certain important issues. The first is the window size. A small window may not contain enough examples for successful learning. A big window may contain misleading examples from outdated contexts. Ideally, therefore, the window should grow (no old examples deleted) as long as it can be assumed that the context has not changed. When a change is suspected, a certain number of the oldest examples are deleted.

This leads us to the next question: how do we recognize that a context has changed? A simple solution relies on the feedback from the classifier's behavior: the change of context is betrayed by reduced classification performance.

Finally, there is the question of how many of the oldest examples to delete. This depends on the speed of the context change and also on the extent of this change. At one extreme, an abrupt and serious change calls for the deletion of *all* examples. At the other, a very slow transition between two very similar contexts will necessitate the deletion of only a few of the oldest examples.

11.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Suggest examples of domains where the meaning of a given class varies in time and/or geographical location. Suggest examples of domains where previous meanings recur in time.
- Describe the scenario that relies on a stream of time-ordered examples. Explain the principle of the sliding window.
- Discuss the basic engineering issues to be considered in the sliding-window approach.

11.5 Unknown Attribute Values

In many realistic applications, the values of certain attributes are not known. A patient refused to give his age, a measurement device failed, and some information got lost—or is unavailable for other reasons. The consequence is an imperfect training set such as the one shown in Table 11.1, where some values have been replaced with question marks. The learning task becomes complicated if the question marks represent a considerable percentage of all attribute-value fields. The engineer needs to know what kind of damage the unknown values may cause and what solutions to adopt.

Table 11.1 Training examples with missing attribute values

		Crust		Filling			
Example	Shape	Size	Shade	Size	Shade	Weight	Class
Ex1	Circle	Thick	Gray	Thick	Dark	7	pos
Ex2	Circle	Thick	White	Thick	Dark	2	pos
Ex3	Triangle	Thick	Dark	Thick	Gray	2	pos
Ex4	Circle	Thin	White	?	Dark	3	pos
Ex5	Square	Thick	Dark	?	White	4	pos
Ex6	Circle	Thick	White	Thin	Dark	?	pos
Ex7	Circle	Thick	Gray	Thick	White	6	neg
Ex8	Square	Thick	?	Thick	Gray	5	neg
Ex9	Triangle	Thin	Gray	Thin	Dark	5	neg
Ex10	Circle	Thick	Dark	Thick	?	?	neg
Ex11	Square	Thick	White	Thick	Dark	9	neg
Ex12	Triangle	Thick	White	Thick	Gray	8	neg

Negative Effects of Unknown Values In the k-NN classifier, the distance between two vectors can only be calculated if all values are known. True, one can modify the distance metric in a way that quantifies the distance between, say, `red` and `unknown`, but distances calculated in this manner tend to be rather *ad hoc*.

Linear and polynomial classifiers face similar difficulties. Without the knowledge of all attribute values, it is impossible to calculate the weighted sum, $\Sigma w_i x_i$, whose sign tells the classifier which class to choose. Unknown attribute values complicate also the use of Bayesian classifiers and neural networks.

Decision trees are more flexible. In the course of concrete classification, it may happen that the attribute whose value is not known will not find itself on the path from the root node to the terminal node and can thus be ignored.

Trivial Ways to Fill In the Missing Values In the case of a large training set with only a few question marks, no harm is done if the examples with unknown values are simply removed. However, this is impractical in domains where the question marks are ubiquitous.

In domains of this kind, we may have to replace the question marks with at least some values, even if incorrect or imprecise ones. In the case of a discrete attribute, the question mark can be replaced with the attribute's most common value. Thus in example $\texttt{ex}_8$ in Table 11.1, the unknown `crust-shade` will be deemed `white` because this is this attribute's most frequent value in the given training set. In the case of a continuous-valued attribute, the average value can be used. In $\texttt{ex}_6$ and $\texttt{ex}_{10}$, the value of `weight` is unknown. Among the 10 examples where it *is* known, the average is `weight=5.1`, and this is what we will use in $\texttt{ex}_6$ and $\texttt{ex}_{10}$.

Of course, replacing the question marks with the most frequent or average values renders the examples' description somewhat dubious. When *many* values are missing, more sophisticated methods are needed.

Learning to Guess the Missing Values The most common values and average values can be misleading. A better method to replace the question marks relies on the fact that attributes are rarely independent from each other. For instance, the taller the man, the greater his body-weight. If the `weight` of someone with `height=6.5` feet is unknown, it would be foolish to use the average weight from the entire population; the tall guy is likely to be heavier than that. We will do better calculating the average weight among those with, say, `height` > 6.

The last argument considered a pair of mutually dependent attributes. Quite often, however, the interrelations will involve three or more attributes. The engineer may then want to employ a mechanism to predict the unknown values with the help of machine learning. A pseudo-code of a simple possibility is given in Table 11.2.

Here is the principle. Suppose that *at* is an attribute that, in the original training set, T, has many question marks which we want to replace with concrete values. We convert the original training set, T, to a new training set, T', where the original class (e.g., `pos` or `neg`) is treated as just another attribute, whereas *at* now becomes the class. Next, we remove all examples whose values of *at* are unknown. From the rest, we induce a classifier (e.g., a decision tree) and then use this classifier to supply the missing values.

Table 11.2 Algorithm to replace question marks with concrete attribute values

```
Let $T$ be the original training set
Let $at$ be the attribute with unknown values

1. Create a new training set, $T'$, in which $at$ becomes the class label; the examples are described
   by all the remaining attributes, the former class label (e.g., pos versus neg) being treated like
   just another attribute.
2. Remove from $T'$ all examples in which the value of $at$ is unknown. This results in yet another
   training set, $T''$.
3. From $T''$, induce a classifier.
4. Using classifier $C$, determine the values of $at$ in those examples in $T$ where its values are
   unknown
```

11.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What are the main difficulties caused by unknown attribute values? What are the consequences in different machine-learning paradigms?
- Describe the trivial methods of dealing with unknown attribute values. Discuss their shortcomings.
- Explain how to use machine learning when seeking to supply the unknown attribute values.

11.6 Attribute Selection

In some domains, such as text categorization from Sect. 8.6, examples are described by a great many attributes: tens of thousands or even more. Learning from data sources of this kind can be prohibitively expensive, and the class may not be PAC-learnable anyway, given the huge size of the instance space. Fortunately, many of these attributes are unnecessary and can be removed.

Irrelevant and Redundant Attributes Not all attributes are created equal. Some are irrelevant in the sense that their values do not have any effect on an example's class. Others are redundant because their values can be derived from other attributes in the way that `age` can be obtained from `current-date` and `date-of-birth`. Irrelevant and redundant attributes tend to mislead machine learning, for instance, by distorting the vector-to-vector distances used in k-NN classifiers. Other paradigms, such as decision trees, are less vulnerable, but they may suffer from unacceptable computational costs.

Extremely Long Attribute Vectors Text categorization relies on tens of thousands of attributes, which imposes an obvious limit on learnability. The induced classifiers are prone to overfit the training data and disappoint on future examples. Also the computational costs may grow too high; for instance, each additional attribute increases the number of weights to be trained in a multilayer perceptron. Finally, examples described by extremely long vectors are sparse, which may negatively affect, say, k-NN classifiers. For all these reasons, the removal of unnecessary attributes is more than desirable.

Trial and Error in Attribute Selection Suppose we have induced a classifier, say, a multilayer perceptron. The importance of the individual attributes can be assessed in the following manner.

Remove one input signal (attribute), including all the links leading from it to the hidden layer, and observe the average mean squared error (MSE) on the training set. Repeat this for each of the remaining attributes, and decide which of these removals has resulted in the smallest increase of the MSE. If this increase appears acceptable, remove the attribute and repeat the procedure recursively with the remaining attributes.

Something similar can be implemented in other paradigms such as the k-NN approach, Bayesian classifiers, linear classifiers, and RBF networks. In each of them, less important attributes can thus be identified; unfortunately, this is done only after the classifier has been induced.

Filtering by Information Gain A simple method of attribute selection prior to classifier induction is *filtering*. The idea is to decide how useful the individual attributes are likely to be for the machine-learning task at hand and then keep only the best. How many to keep and how many to discard are decided by trial and error.

In the case of discrete attributes, the *information gain* from Sect. 5.3 can be used, and the binarization technique from Sect. 5.4 makes information gain applicable even for continuous-valued attributes.

Typical Observation Figure 11.4 shows a typical observation. If we order the attributes from left to right according to their information gains, we will notice that some attributes (perhaps just small percentage) are marked by high information

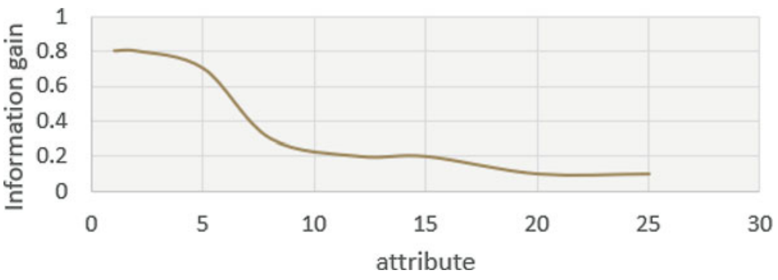


Fig. 11.4 One simple approach to attribute filters. Vertical axis represents information gain. Horizontal axis represents attributes, in descending order of their attribute gain

gain, but others are irrelevant, which is reflected by the sudden drop in the curve in the graph.

Graphical representation of this kind makes it easy to decide where to draw the line between the attributes to be kept and those to be discarded. One can also write a little program to identify the location of the drop.

Criticism of the Filtering Approach Filtering ignores relations between attributes. Also, it is not good at discovering redundancy. Suppose that some attribute A can be derived from attribute B, perhaps by a function such as $A = 3 \times B$. In that event, both attributes provide the same information gain. As such, they will be sitting next to each other in the graph from Fig. 11.4, but it will not be immediately obvious that one of them is redundant.

WINNOW and Decision Trees For the needs of filtering, we can take advantage of some of the techniques we know from the previous chapters. For instance, the reader will recall that the algorithm WINNOW from Sect. 4.3 significantly reduced the weights of irrelevant attributes, and indeed this approach is to be recommended, especially in view of its low computational costs. However, WINNOW suffers from the weakness that the previous paragraph complained about: it fails to detect redundancy.

Another possibility is to induce a decision tree and then use only those attributes that appear in the tests in the induced tree's internal nodes. The reader will recall that this approach was used in some of the simple applications discussed in Chap. 8. The advantage is that decision trees are good at eliminating not only irrelevant attributes but also redundant attributes.

Wrapper Approach More powerful, though computationally more expensive, is the *wrapper* approach. Here is the principle. Suppose we want to compare the utility of two attribute sets, A_1 and A_2. From the original training set, T, we create two training sets, T_1 and T_2. Both contain the same examples as T, but T_1 describes the examples with attributes from A_1, whereas T_2 uses attributes from A_2. From the two newly created training subsets, two classifiers are induced and evaluated on some independent data, T_E. The attribute set that results in the higher classification performance is then preferred.

The quality of a given attribute set is therefore assessed by the success of a classifier induced using these attributes. This seems reasonable—but only until we realize that it may be impractical to experiment with every single subset of attributes.

Sequential Attribute Selection The pseudo-code in Table 11.3 represents a technique that is computationally more acceptable than the plain *wrapper* approach. The input is the training set, T, and a set of attributes, A. The output is a subset, $S \in A$, of attributes that can be regarded as useful.

At the beginning, S is empty. At each step, the technique chooses the best attribute from A and adds it to S. The meaning of "best" is here determined by the classification performance (on an independent testing set) of the classifier induced from examples described by the attributes from S. The algorithm stops if no addition

Table 11.3 Sequential attribute selection by the *wrapper* approach

```
Divide the available set of pre-classified examples into two parts, $T_{train}$ and $T_{test}$. Let $A$ be the
set of all attributes, and let $S$ be an empty set

1. For every attribute, $at_i \in A$:
     (i) add $at_i$ to $S$; describe all examples in $T_{train}$ and $T_{test}$ by attributes from $S$;
    (ii) induce a classifier from $T_{train}$, and then evaluate its performance on $T_{test}$; denote this
         performance by $p_i$;
   (iii) remove $at_i$ from $S$.
2. Identify the attribute that resulted in the highest value of $p_i$. Remove this attribute from $A$, and
   add it to $S$.
3. If $A = \emptyset$, stop; if the latest attribute addition did not improve performance, remove this last
   attribute from $S$ and stop, too. In either case, $S$ is the final set of attributes.
4. If the previous step did not stop the program, return to step 1.
```

to S leads to an improvement of the classification performance or if there are no more attributes to be added to S.

Wrappers or Filters? Each of the two approaches is its strengths and weaknesses, and there is no golden rule to tell us which of them to prefer. Generally speaking, *wrappers* are capable of identifying very good attribute subsets, but often at almost prohibitive computational costs (especially in domains with a great many attributes).

Filters are computationally more efficient but often fail to identify redundant attributes. Moreover, they tend to ignore the fact that some attributes are meaningful only in combination with other attributes. Of course, there is always the possibility to combine the two approached. In the first step, a filter is used to eliminate attributes that are clearly useless; in the second, the wrapper approach is applied to this much smaller set of attributes.

11.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- In what sense do we say that some attributes are less useful than others? Why is the engineer often incapable of choosing the best attributes beforehand?
- How can irrelevant or redundant attributes be identified based on the classification behavior of the induced classifier?
- Explain the principles of the filter-based approaches discussed in this section. What are their strengths and weaknesses?
- Describe the principle of the *wrapper* technique, and explain its use in sequential attribute selection. What are its strengths and weaknesses?

11.7 Miscellaneous

Certain practical issues encountered in machine learning do not merit a separate section, and yet they should not be ignored. Let us take a look at some of them.

Synthesis of Higher-Level Features Earlier chapters of this book repeatedly complained that the attributes describing the examples often of a very low level. More than once did the text suggest that perhaps some techniques should be employed to create higher-level features as functions of the original ones.

Upon reflection, we realize that this is accomplished by multilayer perceptrons where the new features are defined as the outputs of hidden-layer neurons. The same can be said about the Gaussian outputs of RBF networks. Also, each branch of a decision tree defines a feature that is a conjunction of attribute tests. Chapter 15 will present the technique of *auto-encoding* where the creation of new features is one of the main objects. The ability to transform the example description into something meaningful is perhaps the main secret behind the successes of deep learning (see Chap. 16).

Linearly Ordered Classes In some domains, each example is labeled with one out of several (perhaps many) classes that can be ordered. For instance, suppose that the output class is `month`, with values `january` through `december`. When measuring the performance of the induced classifier, it can be misleading to assume that confusing `june` for `may` is an error of the same magnitude as misclassifying `june` for `december`.[2]

Not only during performance evaluation but also in the course of the induction process, attention should therefore be paid to the way the classes are ordered. One possibility is to begin by grouping neighboring class labels, say, into `spring`, `summer`, `fall`, and `winter`. After the induction of a classifier for each group, the next step may proceed to the recognition of each month within its season.

Regression Instead of Classification In some applications, the expected output is not a discrete-valued class, but rather a number from a continuous range. For instance, this can be the case when the software is to predict a value of a stock-market index. These kinds of problems are called *regression*. In this book, we do not address them. The simplest way to deal with regression within the framework of machine learning is to replace the continuum with sub-intervals and then treat each sub-interval as a separate class. Note that the task would then belong to the category of "classes that can be linearly ordered" mentioned in the previous paragraph.

Some of the paradigms from the previous chapters can deal with regression directly. For instance, this is the case of the k-NN approach. Instead of being labeled with a class, each training example is provided with the value of the output variable.

[2]The attentive reader will recall that something similar is the case in the sleep classification domain from Sect. 8.4.

Once the k nearest neighbors have been identified, the average value of their output variables is calculated and returned to the user.

Multilayer perceptrons are very good at the regression task because the outputs of the output-layer neurons are values from continuous domains. If the `sigmoid` activation is used, the outputs are constrained to the (90, 1) interval, and it is thus better to use for the activation function either ReLU or LReLU. Also RBF networks can be adapted for regression without major difficulties.

Ranking Examples of a Given Class This book has ignored one specific version of the class recognition task. In the traditional problem statement, one possible scenario assumes that the user presents a class label, and the classifier returns all examples belonging to this class. Besides this, however, a stronger requirement can be suggested: the classifier may be asked to rank the returned examples according to their relevance to the class.

The reader will find it easy to modify some of the classical techniques in a way that helps them fulfill this goal. For instance, the Bayesian classifier returns for each example the probability that it belongs to the given class; this probability then defines the ranking. The same can be done in the case of a multilayer perceptron whose outputs have been subjected to the *soft-max* function. On the other hand, decision trees of the baseline version from Chap. 5 are not good at this kind of ranking.

Lack of Information in Training Data Suppose you are asked to induce a classifier from a training set that was created by a random-number generator: all attribute values are random, and so are the class labels. Obviously, there is no regularity in such data—and yet machine-learning techniques may induce a classifier with zero error rate on the training set. Of course, this perfect behavior will not translate into similar performance on future examples.

The observation suggests a simple mechanism to decide whether the available data provide the information needed of learning. Divide the data into two subsets, training and testing; the classifier is induced from the former and then applied to both of them. In data that do not contain the information, we will observe small error rate on the training set and high error rate on the testing set. Conversely, the more useful the data, the smaller the difference between the classifiers' performance on the training set and the testing set.

11.7.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain how to use machine learning when measuring the utility of the given data. Suggest examples of domains where this utility is low.
- Suggest a simple approach to deal with domains where the classes can be ordered.
- What is *regression* and how can machine learning address this task?

11.8 Summary and Historical Remarks

- Chapter 7 explained the mathematical arguments behind the slogan, "there is no learning without bias." Practical considerations have now convinced us that bias is in machine learning really important.
- Sometimes, the meaning of the underlying class depends on a context. The context can change in time, in which case we deal with time-varying classes and *concept drift*.
- Classical machine-learning techniques assume that all classes are equally represented in the training set. Quite often, however, this requirement is not satisfied, and the engineer has to deal with the difficulties caused by the problem of *imbalanced training sets*.
- There are three fundamental approaches to the problem of *imbalanced training sets*: modification of the induced classifiers, majority-class undersampling, and minority-class oversampling.
- In many training sets, some attribute values are unknown, and this is an obstacle for certain induction techniques. One possible solution is to use (in place of the unknown values) the most frequent or the average values of the given attributes. More sophisticated solutions employ mechanisms that learn how to fill in the missing values.
- Quite often, the engineer is faced with the necessity to select the most appropriate subset of attributes. Two fundamental approaches can be used: the *filtering* and the *wrapper* techniques.
- Apart from attribute selection, equally important is another task: synthesis of higher-level features as functions of the original attributes.
- In domains with more than two classes, the individual classes can sometimes be ordered. This circumstance can affect performance evaluation. For instance, if the task is to recognize a concrete month, then it is not the same thing if the classifier's output missed the target by 1 month or by five. Even the learning procedure should then perhaps be modified accordingly.
- Sometimes, the output is not a discrete-valued class, but rather a value from a continuous range. This type of problem is called *regression*. This book does not address regression explicitly.

Historical Remarks The idea to distinguish different biases in machine learning was pioneered by Gordon and desJardin (1995). The principle of lifelong learning was pioneered by Thrun and Mitchell (1995). The early influential papers addressing the issue of context were published by Turney (1993) and Katz et al. (1990). An early approach to induction of time-varying concepts was introduced by Kubat (1989), and some early algorithms were described by Widmer and Kubat (1996). The meta-learning approach to context recognition was first suggested by Widmer (1997). The Wrapper approach to attribute selection is introduced by Kohavi (1997).

11.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

11.9.1 Exercises

1. Consider the training set in Table 11.4. How will you replace the missing values (question marks) with the most frequent values? How will you use a decision tree to this end?
2. Once you have replaced the question marks in Table 11.4 with concrete values, identify the two attributes that offer the highest information gain.

11.9.2 Give It Some Thought

1. The text emphasized the difference between two basic sources of error: those caused by the wrong bias (representational or procedural), and those caused by variance in the training data. Suggest an experimental procedure that would give the engineer an idea about how much of the overall error rate in the given application is due to either of these two sources.

Table 11.4 Simple exercise in "unknown values"

		Crust		Filling		
Example	Shape	Size	Shade	Size	Shade	Class
ex_1	Circle	Thick	Gray	Thick	Dark	pos
ex_2	Circle	Thick	White	Thick	Dark	pos
ex_3	Triangle	Thick	Dark	?	Gray	pos
ex_4	Circle	Thin	White	Thin	?	pos
ex_5	Square	Thick	Dark	Thin	White	pos
ex_6	Circle	Thick	White	Thin	Dark	pos
ex_7	Circle	Thick	Gray	Thick	White	neg
ex_8	Square	Thick	White	Thick	Gray	neg
ex_9	Triangle	Thin	Gray	Thin	Dark	neg
ex_{10}	Circle	Thick	Dark	Thick	White	neg
ex_{11}	Square	Thick	White	Thick	Dark	neg
ex_{12}	Triangle	?	White	Thick	Gray	neg

2. Boosting algorithms are known to be quite robust in the face of variance-based errors. Explain why it is so. Further on, *non-homogeneous boosting* from Sect. 9.4 is known to reduce bias-related errors. Again, explain why.
3. Suppose you are going to work with a two-class domain where examples from one class heavily outnumber those from the other class. Will Bayesian classifier be as sensitive to this situation as the nearest-neighbor approach? Support your answer by concrete arguments and suggest experimental verification.
4. This chapter explored the problem of imbalanced training sets within the framework of two-class domains. How does the issue generalize to domains that have more than two classes? Suggest concrete situations where imbalanced classes in multi-class domains can pose a problem.
5. Consider the case of linearly ordered classes mentioned in Sect. 11.7. Using the hint provided in the text, suggest a machine-learning scenario addressing this issue.

11.9.3 Computer Assignments

1. Write a computer program that accepts as input a training set with missing attribute values and outputs an improved training set where the missing values have been replaced with most frequent or average values. Write a computer program that will experimentally ascertain whether this replacement helps or harms the performance of a decision tree induced from such data.
2. Choose some public-domain data, for instance, from the UCI repository.[3] Make sure that this domain has at least one binary attribute. The exercise suggested here will assume that this binary attribute represents a context. Divide the training data into two subsets, each for a different context (a different value of the binary attribute). Then induce from each subset the corresponding context-dependent classifier. Assuming that it is at each time clear which of the two classifiers to use, how much will the average performance of these two classifiers be better than that of a "universal" classifier that has been induced from the entire original training set?

[3] www.ics.uci.edu/~mlearn/MLRepository.html.

Chapter 12
Performance Evaluation

The strategy for performance evaluation in the previous chapters was simple: providing the induced classifier with a set of testing examples with known class labels and then calculating the classifier's error rate on these examples. In reality, however, error rate rarely paints the whole picture, and there are situations in which it is outright misleading. The reader needs to be acquainted with performance criteria that offer a more plastic view of the classifier's behavior.

The experimenter also has to follow proper methodology. Dividing the set of pre-classified examples into two random subsets, one for induction and the other for testing, is impractical if the training set is small because the subsets may not be sufficiently representative. For more reliable results, the experiments need to be repeated in an organized manner: for instance, by stratified sub-sampling or N-fold cross-validation.

This chapter presents criteria for performance evaluation and explains useful experimental strategies. It also discusses other aspects of performance such as learning curves and computational costs. One section warns against typical experimental blunders. Methods of statistical evaluation will be treated in the next chapter.

12.1 Basic Performance Criteria

Let us begin by a formal definition of error rate and classification accuracy. After this, we will discuss the consequences of the rejection of examples for which the evidence supporting the winning class is inadequate.

Correct and Incorrect Classification in Two-Class Domains When testing a classifier on an example whose real class is known, the following outcomes are possible: (1) the example is positive and the classifier correctly recognizes it as such (*true positive*), (2) the example is negative and the classifier correctly recognizes it as such (*true negative*), (3) the example is positive, but the classifier labels it as

© Springer Nature Switzerland AG 2021
M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_12

Table 12.1 Basic quantities for performance evaluation. For instance, N_{FP} is the number of *false positives*: negative examples misclassified as positive

		Labels returned by the classifier	
		pos	neg
True labels:	pos	N_{TP}	N_{FN}
	neg	N_{FP}	N_{TN}

negative (*false negative*), and (4) the example is negative, but the classifier labels it as positive (*false positive*).

When applying the classifier to a set of examples whose real classes are known, each of these four outcomes will occur a different number of times—and these numbers are then employed in the performance criteria defined below. The symbols for the four outcomes are summarized in Table 12.1. Specifically, N_{TP} is the number of *true positives*, N_{TN} is the number of *true negatives*, N_{FP} is the number of *false positives*, and N_{FN} is the number of *false negatives*. In the set T of examples, only these four categories are possible; therefore, the size of the set, $|T|$, is $|T| = N_{FP} + N_{FN} + N_{TP} + N_{TN}$.

Correct classifications are either *true positives* or *true negatives*; the number of correct classification is thus $N_{TP} + N_{TN}$. On the other hand, errors are either *false positives* or *false negatives*; the number of errors is thus $N_{FP} + N_{FN}$.

Error Rate and Classification Accuracy A classifier's *error rate*, E, is the frequency of errors made in a set of examples. It is calculated by dividing the number of errors, $N_{FP}+N_{FN}$, by the total number of examples, $N_{TP}+N_{TN}+N_{FP}+N_{FN}$:

$$E = \frac{N_{FP} + N_{FN}}{N_{FP} + N_{FN} + N_{TP} + N_{TN}} \tag{12.1}$$

Sometimes, the engineer prefers the complementary quantity, *classification accuracy*, Acc: the frequency of correct classifications in a set of examples. Classification accuracy is calculated by dividing the number of correct classifications, $N_{TP}+N_{TN}$, by the total number of examples. Note that $Acc = 1 - E$.

$$Acc = \frac{N_{TP} + N_{TN}}{N_{FP} + N_{FN} + N_{TP} + N_{TN}} \tag{12.2}$$

Rejecting an Example may be Better than Misclassifying it In the context of character recognition, Sect. 8.2 suggested that the classifier should be allowed not to classify an example if the evidence supporting the winning class is not strong enough. The motivation is simple: in some domains, the penalty for misclassification is much higher than the penalty for not making any classification at all.

An illustration is easy to find. The consequence of a classifier's refusal to return the value of the postal code is that the packet's destination has to be determined by a human operator. To be sure, manual processing is more expensive than automatic processing, but perhaps not prohibitively so—whereas the classifier's error results in the packet being sent to a wrong place, causing a serious delay in delivery, and this can be much more costly. Similarly, an incorrect medical diagnosis can be more expensive than no diagnosis at all. Lack of knowledge can be remedied by additional tests, but the wrong diagnosis may result in a harmful treatment.

How to Decide when to Reject The classifier should therefore be allowed to refuse to classify an example if the evidence is insufficient. In some machine-learning paradigms, the term *insufficient evidence* is easy to define. Suppose that, in a 7-NN classifier, four neighbors favor the positive class, and the remaining three favor the negative class. The final count being four versus three, the case is "too close to call." More generally, the engineer may define a threshold for the minimum difference between the number of votes favoring the winning class and those favoring the runner-up.

In Bayesian classifiers, this is easily implemented, as well. If the difference between the probabilities of the two most strongly supported classes falls short of a user-specified minimum (say, 0.55 for `pos` versus 0.45 for `neg`), the example is rejected as ambiguous. Something similar is possible also in neural networks: compare the signals returned by the corresponding output neurons, and refuse to classify if there is no clear-cut winner.

In other paradigms, such as decision trees, the rejection mechanism cannot be implemented without certain "additional tricks."

Advantages and Disadvantages of Example Rejection The classifier that occasionally refuses to make a decision is of course less likely to go wrong. Its error rate will thus be lower, and the more examples are rejected, the lower the error rate. There are some limits, though. It may seem a good thing to see the error rate go down almost to zero, but not if it means that most of the examples are rejected. A classifier that never classifies is not very useful. Which of these two aspects (low error rate versus frequent rejections) is more important depends on the concrete circumstances of the given application.

Figure 12.1 illustrates the trade-offs involved in these decisions. The horizontal axis represents a parameter capable of adjusting the rejection rate. As we move from left to right, the rejection rate increases, whereas the error rate goes down until, at the extreme, all examples are rejected. At this point, the zero error rate is a poor consolation for having a classifier that never classifies. Here is the verdict: occasional rejection of unclear examples makes a lot of sense, but the rejection rate should be kept within reasonable bounds.

12.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Define the terms *false negative, false positive, true negative*, and *true positive*.
- Write down the formulas for classification accuracy and error rate. How are the two criteria interrelated?

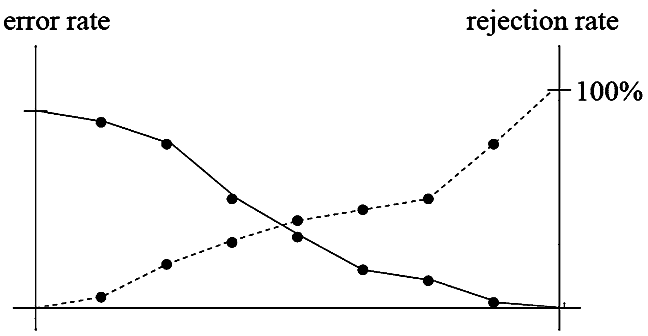


Fig. 12.1 Error rate can be reduced by allowing the classifier to refuse to classify an example if available evidence is weak. Error rate drops to $E = 0$ when all examples are rejected—but this is not what we want

- When should a classifier refuse to classify an example? How would you implement this feature in concrete classifiers?
- How does rejection rate relate to error rate? What is the trade-off between the two? Under what circumstances is it good that the classifier refuses to classify, and under what circumstances is it harmful?

12.2 Precision and Recall

In imbalanced domains, examples of one class outnumber examples of the other class. In this event, error rate can be misleading. Consider the case where only 2% of all examples are positive, and all the remaining 98% are negative. A classifier that returns the negative class for every one of these examples will be correct 98% of the time, which means error rate of only 2%. This looks like an impressive feat, but the user will hardly be happy about a classifier that never recognizes a positive example.

Imbalanced Classes Revisited This observation is worth remembering because domains with imbalanced classes are quite common. We encountered some of them in Chaps. 8 and 11, and many others can be found. Thus in automated information retrieval, the user may want a scientific document dealing with, say, "performance evaluation of classifiers." No matter how attractive the topic may appear to this person, relevant papers represent only a tiny fraction of the millions of available documents. Likewise, patients suffering from a specific disease tend to be rare in the population, and the same goes for the occurrence of defaults on mortgage payments or fraudulent uses of credit cards. The author is tempted to claim that most realistic applications are in some degree marked by imbalanced classes.

In domains of this kind, error rate and classification accuracy do not tell us much about the classifier's practical utility. Rather than averaging the performance over both (or all) classes, we need criteria that focus on a class which, while important, is represented by only a few examples. Let us take a look at some of them.

Precision By this we mean the percentage of true positives, N_{TP}, among all examples that the classifier has labeled as positive: $N_{TP} + N_{FP}$. The value is obtained by the following formula:

$$Pr = \frac{N_{TP}}{N_{TP} + N_{FP}} \tag{12.3}$$

Put another way, *precision* is the probability that the classifier is right when labeling an example as positive.

Recall By this we mean the probability that a positive example will be correctly recognized by the classifier. The value is obtained by dividing the number of true positives, N_{TP}, by the number of positives in the given set: $N_{TP} + N_{FN}$:

$$Re = \frac{N_{TP}}{N_{TP} + N_{FN}} \tag{12.4}$$

Note that the last two formulas differ only in the denominator. This makes sense. Whereas *precision* is the frequency of true positives among all examples deemed positive by the classifier, *recall* is the frequency of true positives among all positive examples in the set.

Illustration of the Two Criteria Table 12.2 illustrates the behavior of the two criteria in a simple domain with an imbalanced representation of two classes, positive and negative. The induced classifier, while exhibiting an impressive classification accuracy, suffers from poor *precision* and *recall*. Specifically, *precision* of $Pr = 0.40$ means that of the fifty examples labeled as positive by the classifier, only twenty are indeed positive, the remaining thirty being false positives. With *recall*, things are even worse: out of the seventy positive examples in the testing set, only twenty were correctly identified as such by the classifier.

Suppose that the engineer decides to improve the situation by tweaking some of the classifier's internal parameters, and suppose that this results in an increased number of true positives (from $N_{TP} = 20$ to $N_{TP} = 30$) and a drop in the number of false positives (from $N_{FP} = 30$ to $N_{FP} = 20$). As a result, the number of false negatives has dropped from $N_{FN} = 50$ to $N_{FN} = 40$. The calculations in Table 12.2 indicate that both *precision* and *recall* improved considerably from $Pr = 0.40$ to $Pr = 0.60$ and from $Re = 0.29$ to $Re = 0.43$, whereas classification accuracy remained virtually unchanged.

When High Precision Matters In some domains, *precision* is more important than *recall*. For instance, when you make a purchase at a web-based company, their recommender system often informs you that, "Customers purchasing X buy also Y." The intention is to cajole you into buying Y as well.

Table 12.2 Illustration of the two criteria: *precision* and *recall*

Suppose a classifier has been induced. Evaluation on a testing set resulted in the numbers summarized in the following table:

		Labels returned by the classifier	
		pos	neg
True labels	pos	20	50
	neg	30	900

From these, the values of precision, recall, and accuracy are obtained:

$$\text{precision} = \frac{20}{50} = 0.40; \quad \text{recall} = \frac{20}{70} = 0.29; \quad \text{accuracy} = \frac{920}{1000} = 0.92$$

Suppose the classifier's parameters were modified with the intention to improve its behavior on positive examples. After the modification, evaluation on a testing set resulted in the following numbers.

		Labels returned by the classifier	
		pos	neg
True labels	pos	30	40
	neg	20	910

From these, the values of precision, recall, and accuracy are obtained:

$$\text{precision} = \frac{30}{50} = 0.60; \quad \text{recall} = \frac{30}{70} = 0.43; \quad \text{accuracy} = \frac{940}{1000} = 0.94$$

The reader can see that both precision and recall have improved considerably, whereas classification accuracy has improved only marginally.

Recommender systems are often created by machine learning applied to the company's historical records.[1] When evaluating their performance, the engineer wants to achieve high *precision*: the customers better be happy about the recommended merchandise or else they will ignore the recommendations in the future.

On the other hand, *recall* is here unimportant. The list of the recommended items has to be short, and so it does not matter much that the system identifies only a small percentage of the items that the customers may like.

When High Recall Matters In other domains, *recall* can be much more important. This is often the case in medical diagnosis. A patient suffering from X, and properly diagnosed as such, represents a true positive. A patient suffering from X but *not* diagnosed as such represents a false negative, something the doctor wants to avoid—which means that N_{FN} should be small. In the definition of *recall*, $Re = \frac{N_{TP}}{N_{TP}+N_{FN}}$,

[1]The concrete techniques employed to this end are more advanced than those discussed in this textbook and therefore are not treated here.

the number of false negatives appears in the denominator. Consequently, a small value of N_{FN} implies a high value of *recall*.

Trading the Two Types of Error In many classifiers, tweaking certain parameters can modify the values of N_{FP} and N_{FN}, thus affecting the classifier's behavior, for instance, by improving *recall* at the cost of worsened *precision* or vice versa. This can be useful in domains where the user knows which of these two quantities is more important.

In the k-NN classifier, the engineer may insist that an example be labeled as negative unless the evidence for the positive class is really strong. For instance, if four out of the seven nearest neighbors are positive, the classifier may still be instructed to return the negative label (in spite of the small majority in favor of the positive class). This preference for the negative class is likely to reduce the number of false positives, though usually at the price of more frequent false negatives. Even stronger reduction in N_{FP} (at the cost of increased N_{FN}) may be achieved by requesting that any example be deemed negative unless at least five of the seven nearest neighbors are positive.

Something similar can be accomplished also in other machine-learning paradigms such as Bayesian classifiers or neural networks. The idea is to label the example with the preferred class unless really strong evidence suggests otherwise.

ROC Curves The behavior of the classifier under different parameter settings can be visualized by the so-called ROC curve, a graph where the horizontal axis represents error rate on negative examples (the number of false positives among all negative examples) and the vertical axis represents classification accuracy on positive examples (the number of true positives among all positive examples). Figure 12.2 shows the ROC curves of two classifiers, $c1$ and $c2$. Ideally, we would

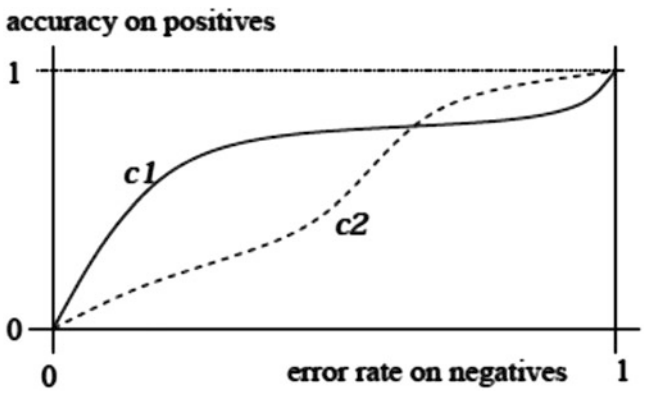


Fig. 12.2 Two ROC curves, $c1$ and $c2$. The numbers of false positives and false negatives are modified by different settings of certain parameters of the classifiers

like to reach the upper-left corner that represents zero error rate on the negatives and 100% accuracy on the positives. This, of course, is rarely possible.

The question to ask is which of the two, $c1$ or $c2$, is better. The answer is possible only if we understand the specific needs of the application at hand. All we can say by just looking at the graph is that $c1$ outperforms $c2$ on the positives in the region with low error rate on the negatives. As the error rate on negative examples increases, $c2$ outperforms $c1$ on the positive examples. Again, whether this is good or bad can only be decided by the user who knows the penalties for different kinds of error—see the earlier discussion of when to prefer *precision* and when *recall*.

12.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- In what kind of data would we rather measure the classifier's performance by *precision* and *recall* instead of by error rate?
- What formulas calculate the values of these criteria? What is the main difference between the two formulas?
- Under what circumstances do we prefer high *precision*, and under what circumstances do we place more emphasis on high *recall*?
- Explain the nature of the ROC curve. What extra information does the curve convey about the classifier's behavior? How does the ROC curve help the user in choosing the classifier?

12.3 Other Ways to Measure Performance

Apart from error rate, classification accuracy, *precision*, and *recall*, other criteria are sometimes used, each reflecting a somewhat different aspect of the classifier's behavior. Let us take a quick look at some of the most common ones.

Combining *Precision* and *Recall* in F_β Working with two different criteria may be inconvenient; it is in the human nature to want to quantify performance by a single number. In the case of *precision* and *recall*, attempts have been made to combine the two. The best-known solution, F_β, is defined by the following formula:

$$F_\beta = \frac{(\beta^2 + 1) \times Pr \times Re}{\beta^2 \times Pr + Re} \tag{12.5}$$

The parameter, $\beta \in [0, \infty)$, enables the user to weigh the relative importance of the two criteria. If $\beta > 1$, more weight is given to *recall*. If $\beta < 1$, more weight is apportioned to *precision*. It would be easy to show that F_β converges to *recall* when $\beta \to \infty$ and to *precision* when $\beta = 0$.

Quite often, the engineer cannot really say which of the two, *precision* or *recall*, is more important and how much more important it is. In that event, he or she chooses to work with the neutral value, $\beta = 1$:

$$F_1 = \frac{2 \times Pr \times Re}{Pr + Re} \tag{12.6}$$

Numeric Example Suppose that the evaluation of a classifier on a testing set resulted in the values summarized in the upper part of Table 12.2. For these, the table provided the values of *precision* and *recall*, respectively, as $Pr = 0.40$ and $Re = 0.29$. Using these numbers, let us calculate F_β for the following values of the parameter: $\beta = 0.2$, $\beta = 1$, and $\beta = 5$.

$$F_{0.2} = \frac{(0.2^2 + 1) \times 0.4 \times 0.29}{0.2^2 \times 0.4 + 0.29} = \frac{0.121}{0.306} = 0.39$$

$$F_1 = \frac{(1^2 + 1) \times 0.4 \times 0.29}{0.4 + 0.29} = \frac{0.232}{0.330} = 0.70$$

$$F_5 = \frac{(5^2 + 1) \times 0.4 \times 0.29}{5^2 \times 0.4 + 0.29} = \frac{3.02}{10.29} = 0.29$$

Sensitivity and Specificity The choice of the performance criterion is often dictated by the application field—with its specific needs and traditions that cannot be ignored. Thus the medical domain is accustomed to assessing performance of their "classifiers" (not necessarily developed by machine learning) by *sensitivity* and *specificity*. In essence, these are nothing but *recall* measured on the positive and negative examples, respectively. Let us be concrete:

Sensitivity is *recall* measured on positive examples:

$$Se = \frac{N_{TP}}{N_{TP} + N_{FN}} \tag{12.7}$$

Note that sensitivity decreases with the growing number of false negatives, N_{FN}, in the formula's denominator. The fewer the false negatives, the higher the sensitivity.
Specificity is *recall* measured on negative examples:

$$Sp = \frac{N_{TN}}{N_{TN} + N_{FP}} \tag{12.8}$$

Note that specificity decreases with the growing number of false positives, N_{FP}, in the formula's denominator. The fewer the false positives, the higher the specificity.

Medical Example Suppose there is a disease X. A test meant to detect X has been applied to a certain population.

Among people suffering from X (the denominator of Eq. 12.7), *sensitivity* gives the percentage of those for whom the disease was confirmed by the positive result of the test. Among healthy people (the denominator of Eq. 12.8), *specificity* gives the percentage of those for whom the negative test result confirmed that they indeed do *not* suffer from X.

Legal Example Modern justice follows the maxim, "It is better to set a guilty one free than to condemn an innocent person." How is this principle interpreted in the terms of the quantities from Table 12.1 and in terms of *sensitivity* and *specificity*?

In the long run, the number of condemned innocents is N_{FP}, and the number of guilty ones set free is N_{FN}. Modern judge wants low N_{FP}, even if N_{FN} becomes high. This means high *specificity* (recall on negative examples), regardless of the possibly low *sensitivity*.

By contrast, a resolute dictator may say, "I want all guilty ones punished, even if it means sentencing some innocents." He thus wants low N_{FN}, even if it comes at the price of high N_{FP}. In other words, he wants high *sensitivity* (recall on positive examples), regardless of possibly low *specificity*.

Gmean When inducing a classifier in a domain with imbalanced class representation, the engineer sometimes wants to see similar performance on both classes, positive and negative. In this event, the geometric mean, gmean, of the two accuracies (on the positive examples and on the negative examples) is used:

$$\texttt{gmean} = \sqrt{acc_{\texttt{pos}} \times acc_{\texttt{neg}}} = \sqrt{\frac{N_{TP}}{N_{TP} + N_{FN}} \times \frac{N_{TN}}{N_{TN} + N_{FP}}} \tag{12.9}$$

Note that *gmean* is the square root of the product of two numbers: *recall* on positive examples and *recall* on negative examples—in other words, the product of *sensitivity* and *specificity*.

Gmean Reflects the Balance Between the Two Values Perhaps the most important aspect of *gmean* is that it depends not only on the concrete values of the two terms under the square root symbol, $acc_{\texttt{pos}}$ and $acc_{\texttt{neg}}$, but also on how close the two values are to each other. A simple numeric example will convince us that this is indeed the case.

Thus the arithmetic average of 0.75 and 0.75 is $(0.75 + 0.75)/2 = 0/75$; also the arithmetic average of 0.55 and 0.95 is $(0.55 + 0.95)/2 = 0.75$. However, the geometric mean of the first pair is $\sqrt{0.75 \times 0.75} = 0.75$, whereas the geometric mean of the second pair is $\sqrt{0.55 \times 0.95} = 0.72$, a smaller number. We can see that *gmean* is indeed smaller when the two numbers are different; the more different they are, the lower the value of *gmean*.

Numeric Example Suppose the evaluation of a classifier on a testing set resulted in the values summarized in the upper part of Table 12.2. The values of *sensitivity*, *specificity*, and *gmean* are calculated as follows:

$$Se = \frac{20}{50+20} = 0.29$$

$$Sp = \frac{900}{900+30} = 0.97$$

$$\mathtt{gmean} = \sqrt{\frac{20}{50+20} \times \frac{900}{900+30}} = \sqrt{0.29 \times 0.97} = 0.53$$

Cost Functions We already know that, in realistic applications, not all errors carry the same penalty: false positives may be more costly than false negatives or the other way round. Worse still, the costs associated with the two types of error may not even be measurable in the same—or at least comparable—units.

Recall the oil-spill-recognition domain discussed in Chap. 8. A false positive here means that a "look-alike" is incorrectly taken for an oil spill (false positive). When this happens, an aircraft is unnecessarily dispatched to verify the case. The costs incurred by this error are those associated with the flight. By contrast, a false negative means that potential environmental hazard has gone undetected, something difficult to cast in monetary terms.

Under such circumstances, the engineer has to be very careful about how to measure the success or failure of the induced classifier. Mathematical formulas can be misleading.

12.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain how F_β combines *precision* and *recall*. Write down the formula, and discuss how different values of β give more weight either to *precision* or to *recall*. Which value of β gives equal weight to both?
- Write down the formulas for *sensitivity* and *specificity*. Explain their nature, and point out their relation to *recall*. When are the two criteria used?
- Write down the formula for *gmean*. Explain the nature of this quantity, and show its relation to *recall*. When is this criterion used?
- Under what circumstances can the costs associated with false positives be different from the costs associated with false negatives? Suggest a situation where mathematical comparison of the costs is almost impossible.

12.4 Learning Curves and Computational Costs

So far, we have focused on how to evaluate the performance of classification behavior. Somewhat different story is how to evaluate the learning algorithms themselves. Facing a concrete application domain, the engineer asks: How efficient is the selected learning technique computationally? How good are the classifiers it produces? Will better results be achieved by resorting to alternative approaches?

This section discusses the *costs* of learning, measured either (1) by the number of examples needed for good results or (2) by the computation time consumed. Another aspect, namely how to compare the ability to induce a high-performance classifier in two competing approaches, will be the subject of the next section.

Learning Curve When evaluating a human subject's ability to learn to solve a problem, psychologists sometimes rely on a *learning curve*, a notion that machine learning has borrowed.

In our context, the learning curve will plot the classification performance of the induced classifier against the size of the training set used for the induction. Two such curves are shown in Fig. 12.3. The horizontal axis gives the number of training examples, and the vertical represents classification accuracy of the induced classifiers. Usually, though not always, the classification accuracy is evaluated on independent testing examples.

Larger training sets usually promise higher classification performance—unless we have reached a point beyond which no further improvement is possible no matter how many examples we use. Ideally, we want to achieve maximum performance with the smallest possible training set. This is dictated by practical considerations. Training examples can be expensive, and their source can be limited anyway. Besides, the more examples we use, the higher the computational costs.

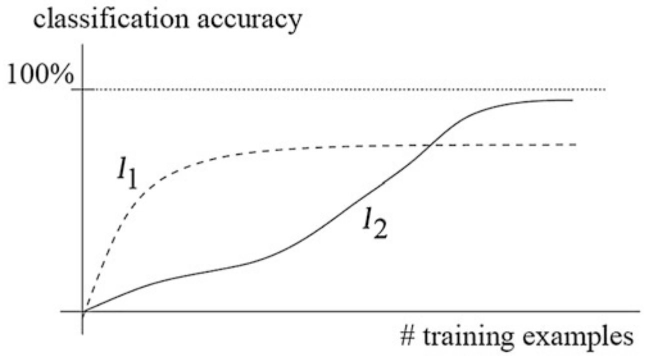


Fig. 12.3 The two learning curves, l_1 and l_2, show how the performances of classifiers induced by competing approaches may depend on the number of training examples

Comparing Learners with Different Learning Curves Figure 12.3 shows the learning curves of two machine-learning programs, l_1 and l_2. The reader can see that the learning curve of the former, l_1, rises very quickly, only to level off at a point beyond which virtually no improvement occurs—perhaps because of the choice of an inappropriate bias (see Sect. 11.1). By contrast, the learning curve of the second program, l_2, does not grow so fast, but in the end it clearly outperforms l_1.

Which of the two curves is preferable depends on the circumstances of the given application. When the source of the training examples is limited, the first learner is clearly to be preferred. If the examples are abundant, the other learner will be deemed more attractive—assuming, of course, that the attendant computational costs are acceptable.

Computational Costs In machine learning, computational costs have two aspects. First, we are concerned about the time consumed by the classifier's induction. Second, we want to make sure that the classifier does not take too long to label a possibly very large set of examples.

The techniques described in this book span a broad spectrum. As for induction costs, the cheapest is the basic version of the k-NN classifier, the only "computation" involved being the necessity to store the training examples.[2] On the other hand, the k-NN classifier has high classification costs. If we have a million training examples, each described by ten thousand attributes, then 10^{10} arithmetic operations will be needed to classify a single example. If we need to classify millions of examples, the costs can run high.

The situation is different in the case of decision trees. These are cheap when used to classify examples: usually only a moderate number of single-attribute tests are needed. However, induction of decision trees can take a lot of time if the training is big and if there are a great many attributes.

Induction and classification costs of other classifiers vary. The engineer needs to understand their nature when choosing the most appropriate approach for the application at hand.

12.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What are the two main aspects of the computational costs considered in machine learning?
- What does the learning curve tell us about an induction algorithm's behavior? What shape of the learning curve represents the ideal? What should we expect in reality?

[2]Some computation will be necessary if we decide to remove noisy or redundant examples or otherwise to pre-process the data.

- Under what circumstances does a steeper learning curve with lower maximum indicate a more favorable situation than a curve that grows more slowly but reaches a higher maximum?

12.5 Methodologies of Experimental Evaluation

The reader understands that different domains favor different paradigms of machine learning. Usually, the choice is not difficult, being guided by the nature of the training data. For instance, if many attributes are suspected to be irrelevant, then a decision tree may be more appropriate than k-NN.

The success of a technique also depends on the settings of its various parameters. True, certain time-tested rules of thumb can often help here; however, the best parameter setting is usually found by experimentation.

Baseline Approach and its Limitations The basic experimental scenario is simple. The set of pre-classified examples is divided into two subsets, one for training and one for testing. The training–testing session is repeated for different parameter settings, and the values that lead to the best performance on the testing set are chosen.

This, however, is realistic only when a great many pre-classified examples are available. In domains where examples are scarce or expensive to obtain, a random division into the training and testing sets will lack objectivity. Either of the two sets can, by mere chance, fail to represent the given domain adequately. Statisticians know that both the training set and the testing set should have more or less the same distribution of examples. In small sets, of course, this requirement is hard to satisfy.

Random Sub-sampling When the store of pre-classified examples is small, the engineer prefers to repeat the procedure several times. In each run, the set of examples is divided into a different pair of training and testing sets. The measured performances are recorded and then averaged. Of course, one has to make sure that the individual data splits are mutually independent.

Once the procedure has been repeated N times (typically, $N = 5$ or $N = 10$), the results are reported in terms of the average classification accuracy and its standard deviation, say, 84.2 ± 0.6. For the calculation of averages and standard deviations, the formulas from Chap. 2 are used: the average, μ, is obtained by Eq. 2.12, and the standard deviation, σ, is the square root of variance, σ^2, which is calculated using Eq. 2.13.

***N*-Fold Cross-Validation** For advanced statistical evaluations, experienced experimenters often prefer the N-fold cross-validation. Figure 12.4 illustrates the principle. To begin with, the set of pre-classified examples is divided into N equally sized (or almost equally sized) subsets, which in the machine-learning jargon are known as "folds."

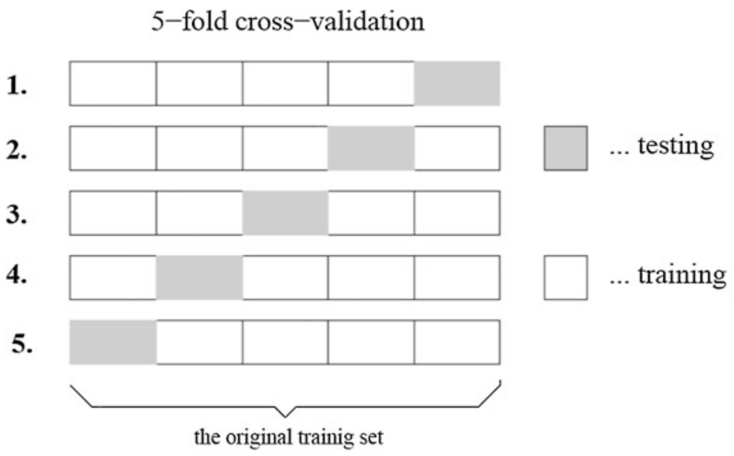


Fig. 12.4 N-fold cross-validation divides the training set into N equally sized subsets. In each of the N experimental runs, a different subset is withheld for testing, and the classifier is induced from the union of the remaining $N-1$ subsets

N-fold cross-validation then runs N experiments. In each, one of the N subsets is removed so as to be used only for testing (this guarantees a different testing set for each run). Training is then run on the union of the remaining $N-1$ subsets. The results are averaged, and the standard deviation is calculated.

The advantage of N-fold cross-validation as compared to random sub-sampling is that the testing sets are disjoint (non-overlapping), which is deemed advantageous for certain types of statistical analysis of the classifiers' reliability (see Chap. 13).

Stratified Approaches Consider a domain with 60 positive and 940 negative examples. If we rely on N-fold cross-validation with $N = 10$, then each of the folds will consist of 100 examples, with probably a very different proportion of positive examples. On average, there will be six positives in each fold, but the concrete numbers will vary; it can even happen that some folds will not contain any single positive example.

This is why the experimenter prefers a so-called *stratified* approach. The idea is to make sure that each of the N folds has approximately the same representation of both classes. For instance, when using the 5-fold cross-validation in a domain with 60 positive and 940 and negative examples, respectively, each fold should contain 200 examples of which 12 are positive.

The same principle is often followed in random sub-sampling (which, admittedly, is in its stratified version no longer totally random). Again, the point is to make sure that each training set, and each testing set, has about the same representation of each class.

5x2 Cross-Validation (5x2CV) There is yet another approach to experimental evaluation of machine-learning techniques, the so-called 5x2 cross-validation,

Table 12.3 Pseudo-code for 5x2 cross-validation (5x2CV)

```
Let $T$ be the original set of pre-classified examples.

(1) Divide $T$ randomly into two equally sized subsets. Repeat the division five times, obtaining
    five pairs of subsets denoted as $T_{i1}$ and $T_{i2}$ (for $i = 1, \ldots, 5$).
(2) For each of these pairs, use $T_{i1}$ for training and $T_{i2}$ for testing, and then the other way round.
(3) For the ten training–testing sessions thus completed, calculate the mean value and the
    standard deviation of the selected performance criterion.
```

sometimes abbreviated as 5x2CV. The principle is built around a combination of random sub-sampling and 2-fold cross-validation.

To be specific, 5x2CV divides the set of pre-classified examples into two equally sized parts, T_1 and T_2. Next, it uses T_1 for training and T_2 for testing, and then the other way around: T_2 for training and T_1 for testing. The procedure is repeated 5 times, each time with a different random division into the two subsets. All in all, ten training–testing sessions are thus conducted and the results averaged. The principle is summarized by the pseudo-code in Table 12.3.

Again, many experimenters prefer to work with the stratified version of this methodology, making sure that the representation of each class is about the same in each of the ten parts used in the experiments.

***No-Free-Lunch* Theorem** It would be foolish to expect some machine-learning technique to be a holy grail, a tool to be preferred under all circumstances. Nothing like this exists. The reader by now understands that each paradigm has its advantages that make it succeed in some domains—and shortcomings that make it fail in others. Only personal experience supported by systematic experimentation tells the engineer which type of classifier and which induction algorithm to choose for the task at hand. The truth is that no machine-learning approach will outperform all other approaches under all circumstances.

Mathematicians have been able to demonstrate the validity of this statement by a rigorous proof. The result is known under the somewhat fancy name, *no-free-lunch theorem*.

12.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the difference between N-fold cross-validation and random sub-sampling? Why do we prefer to employ their stratified versions?
- Explain the principle of the 5x2 cross-validation (5x2CV), including its stratified version.
- What is the *no-free-lunch theorem* telling us?

12.6 Experimental Blunders to Avoid

In their first machine-learning projects, beginners often repeat blunders that can easily be avoided. Let us mention some of the most typical ones.

Treating Synthetic Examples Properly One of the techniques suggested in the context of imbalanced training sets in Sect. 11.3 was minority-class oversampling. The idea was to synthesize additional training examples either as exact copies of existing ones or as their slightly distorted (noisy) variants. The intention was to enrich the training set.

Here is the blunder to avoid. The engineer takes example x and creates its exact copy, x_1. When the random division into the training and testing sets is made, x falls into the training set and x_1 into the testing set. This means that an example used for training is used also for testing—which of course is not a sound method of performance evaluation.

The lesson is clear. First, the division into the training and testing sets should be made, and only after that, the examples within the training set can be multiplied.

***IID* Requirement** Mathematicians figured out long time ago that the independent data on which we want to evaluate the classifier should have the same properties as the training data—and as any data to which the classifier is to be applied in the future. This is known as the *iid* requirement, the acronym standing for independent but identically distributed. The previous paragraph presented a situation where the first requirement was violated: *independence*. A well-known historical anecdote will show how the second one was violated: the need for *identical distribution*.

The first time an opinion poll was conducted during the U.S. presidential elections was in 1932. The results predicted the victory of the republican candidate—but it was the democrat, F.D. Roosevelt, who in the end won. Soon afterward, the cause of the confusion was discovered: the poll was conducted by phone calls; at that time, only well-to-do people had telephones, and well-to-do people tended to favor the republican party. Put in machine-learning terms, the distribution of the two classes, republican and democrat, was different in the training set and in the testing set, the latter being here the entire voting population.

The attentive reader will recall that Sect. 11.4 warned against data originating from different contexts such as when software for spoken-language understanding is trained on London speakers and then applied in New York. This, too, would violate the *iid* requirement.

Failing to Permutate Pre-classified Experimental Data When testing their machine-learning programs, students often rely on publicly available benchmark domains. Such experimentation, however, has to be done with caution. In some of these test-beds, the examples are ordered by classes. For instance, the first 100 examples are from class C_1, the next 100 examples are from class C_2, and the last 100 examples are from class C_3.

Using the first two-thirds for training and the last third for testing means that the training data contain only examples from classes C_1 and C_2, whereas the testing data contain only examples from class C_3. This, of course, is just another blatant violation of the *iid* requirement from the previous paragraph. No wonder that the experimental results are then dismal.

Overfitting the Testing Set Everybody knows that spectacular classification performance on the training set does not guarantee an equally favorable result on independent testing data. The induced classifier may simply overfit the (possibly noisy) training set. What is less commonly appreciated is that one can just as well overfit the testing data.

Here is an example of how it can happen. A long series of experiments has been conducted, each with different values of the learner's parameters. At the end, the solution that appears best on the testing set is chosen. However, the good result on the testing set may still be nothing but a coincidence. Among the many alternatives, the experimenter has chosen the one that only happened to be good on the relatively small testing set; the classifier overfitted the testing set. This blunder is more common than the reader may suspect.

The tell-tale sign that something of this sort is happening is a situation where the performance on the testing set is better than the performance on the training set. Surprisingly, this sometimes happens even when the engineer employs sound experimental strategies such as stratified N-fold cross-validation.

It thus makes sense to keep some portion of the available pre-classified data aside, using them neither for training nor for testing. Only at the very end will the apparently best classifier be double-checked on this "ultimate testing set." This reduces the problem without eliminating it. There is still the danger, even if now smaller, of overfitting the "ultimate testing set."

12.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of the *iid* requirement. Why is it important? Discuss typical examples of the violation of *independence* and examples of *identical distribution*.
- What is meant by "overfitting the testing test"? How can you avoid it in practical experimentation?

12.7 Summary and Historical Remarks

- The basic criterion for classification performance is *error rate*, E, defined as the percentage of misclassified examples in the given set. Complementary to it is *classification accuracy*, $Acc = 1 - E$.

- When the evidence for any class is weak, the classifier may refuse to classify an example in order to avoid costly errors. Rejection rate then becomes yet another aspect of performance evaluation. Higher rejection rate leads to lower error rate; however, the classifier's utility may be questionable if too many examples are rejected.
- Criteria for classification performance can be defined by the counts of true positives, true negatives, false positives, and false negatives. These are denoted by N_{TP}, N_{TN}, N_{FP}, and N_{FN}, respectively.
- In domains with imbalanced class representation, error rate can be misleading. Better picture is provided by *precision* ($Pr = \frac{N_{TP}}{N_{TP}+N_{FP}}$) and *recall* ($Re = \frac{N_{TP}}{N_{TP}+N_{FN}}$).
- Sometimes, *precision* and *recall* are combined into a single criterion, F_β, defined by the following formula:

  $$F_\beta = \frac{(\beta^2 + 1) \times Pr \times Re}{\beta^2 \times Pr + Re}$$

  The value of β determines the relative importance of *precision* ($\beta < 1$) or *recall* ($\beta > 1$). If the two are equally important, we use $\beta = 1$, obtaining the following:

  $$F_1 = \frac{2 \times Pr \times Re}{Pr + Re}$$

- Among other criteria for classification performance, *sensitivity* ($Se = \frac{N_{TP}}{N_{TP}+N_{FN}}$) and *specificity* ($Sp = \frac{N_{TN}}{N_{TN}+N_{FP}}$) are popular. Sometimes, the geometric means, *gmean* of accuracy on the positive examples and accuracy on the negative examples are used.
- Another important aspect to consider is how many training examples are needed for a certain classification performance. Concrete circumstances are visualized by a *learning curve*.
- Also worth the engineer's attention are the computational costs associated with induction and with classification.
- When comparing alternative machine-learning techniques in domains with limited numbers of pre-classified examples, engineers rely on random sub-sampling, N-fold cross-validation, and the 5x2 cross-validation. *Stratified versions* of these techniques ensure that each training (and testing) set has the same proportion of examples from each class.
- An important experimental maxim is known as the *iid* requirement: the testing data have to be independent of the training data, but identically distributed.
- Besides the danger of training-set overfitting, the engineering has to be careful not to overfit the testing data.

Historical Remarks The performance criteria discussed in this chapter are well established in statistical literature and have been used for such a long time that it is difficult to trace their origin. The exception is the relatively recent *gmean* that was proposed to this end by Kubat et al. (1997).

The idea to reject an example if the k-NN classifier cannot rely on a clear majority was put forward by Hellman (1970) and later analyzed by Louizou and Maybank (1987). The principle of 5x2 cross-validation was suggested and experimentally explored by Dietterich (1998). The no-free-lunch theorem was published by Wolpert (1996).

12.8 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

12.8.1 Exercises

1. Suppose that an evaluation of a classifier resulted in the counts listed in the following table:

		Labels Returned by the classifier	
		`pos`	`neg`
True labels	`pos`	50	50
	`neg`	40	850

   Calculate the values of *precision, recall, sensitivity, specificity*, and *gmean*.
2. Using the data from the previous question, calculate F_β for different values of the parameter: $\beta = 0.5$, $\beta = 1$, and $\beta = 2$.
3. Suppose that an evaluation of a machine-learning technique using five-fold cross-validation resulted in the following testing-set error rates:
   $E_{11} = 0.14$, $E_{12} = 0.16$, $E_{13} = 0.10$, $E_{14} = 0.15$, $E_{15} = 0.18$
   $E_{21} = 0.17$, $E_{22} = 0.15$, $E_{23} = 0.12$, $E_{24} = 0.13$, $E_{25} = 0.20$
   Calculate the mean value of the error rate and its standard deviation, σ, using the formulas from Chap. 2 (recall that standard deviation is the square root of variance, σ^2).

12.8.2 Give it Some Thought

1. Suggest a domain where *precision* is much more important than *recall*. Conversely, suggest a domain where *recall* is more important than *precision*. Suggest different illustrations from those mentioned in this chapter.
2. What aspects of the induced classifier's behavior are reflected in *sensitivity* and *specificity*? Suggest circumstances under which these two give a better picture of the classifier's utility than *precision* and *recall*.
3. Suppose you have induced two classifiers: one with very high *precision* and the other with high *recall*. What can be gained from the combination of the two classifiers? How would you implement this combination? Under what circumstances will the idea fail?
4. What may be the advantages and shortcomings of random sub-sampling in comparison with N-fold cross-validation?
5. How will you organize your experiments so as to avoid, or at least mitigate, the danger of testing-set overfitting?

12.8.3 Computer Assignments

1. Assume that a machine-learning experiment resulted in a table where each row represents a testing example. The first column contains the examples' class labels ("1" or "0" for the positive and the negative examples, respectively), and the second column contains the labels suggested by the induced classifier.
   Write a program that calculates *precision*, *recall*, and F_β for a user-specified β. Write a program that calculates the values of the other performance criteria.
2. Suppose that a training set has the form of a matrix where each row represents an example, each column represents an attribute, and the rightmost column contains the class labels.
   Write a program that divides this set randomly into five pairs of equally sized subsets as required by the 5x2 cross-validation technique. Then write another program that creates the subsets in the *stratified* manner where each subset has approximately the same representation of each class.
3. Write a computer program that accepts as input a training set and outputs N subsets to be used in N-fold cross-validation. Make sure that the approach is *stratified*. How will your program have to be modified if you later decide to use the 5x2 cross-validation instead of the plain N-fold cross-validation?

Chapter 13
Statistical Significance

Suppose you have evaluated a classifier's performance on an independent testing set. To what extent can the results be trusted? When a flipped coin comes up *heads* eight times out of ten, any sensible person will say this is nothing but a fluke, easily refuted by new trials. Similar caution is in place when evaluating a classifier in machine learning. To measure its performance on a testing set is not enough; just as important is an estimate of the chances that the obtained value is a reliable estimate. This information can be provided by an informed application of mathematical statistics.

To acquaint the student with the requisite techniques and procedures, this chapter introduces such fundamental concepts as standard error, confidence intervals, and hypothesis testing, explaining and discussing them from the perspective of the needs of machine learning.

13.1 Sampling a Population

If we test a classifier on several different testing sets, the error rate on each of them will be different—but not quite arbitrary: the distribution of the measured values cannot escape the laws of statistics. A good understanding of these laws can help us decide how representative the results of our measurements are.

Observation Table 13.1 contains one hundred zeros and ones obtained from a random-number generator whose parameters have been set to make it return a zero twenty percentage of the time and a one eighty percent of the time. The real percentages in the returned data are of course slightly different than what the setting required. In this particular case, the table contains 82 ones and 18 zeros.

The numbers on the side and at the bottom of the table tell us how many ones are found in each row and column. Based on these, we can say that the proportions of ones in the first two rows are 0.6 and 0.9, respectively, because each row contains

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_13

Table 13.1 A set of binary values returned by a random-number generator adjusted to return a one 80% of the time. In reality, there are 82 ones and 18 zeros. At the ends of the rows and columns are the corresponding sums

0	0	1	0	1	1	1	0	1	1	6
1	1	0	1	1	1	1	1	1	1	9
1	1	1	0	1	1	1	1	1	1	9
1	1	1	1	1	1	0	0	1	1	8
1	1	1	0	1	0	1	0	1	1	7
1	1	1	1	1	1	1	1	1	1	10
1	1	1	1	1	1	1	1	0	1	9
1	1	1	0	1	1	1	0	1	1	8
1	1	1	0	1	0	1	1	1	1	8
1	0	1	1	1	1	1	0	1	1	8
9	8	9	5	10	8	9	5	9	10	82

10 numbers. Likewise, the proportions of ones in the first two columns are 0.9 and 0.8. The average of these four proportions is $(0.6 + 0.9 + 0.9 + 0.8)/4 = 0.80$, and the standard deviation is 0.08.[1]

For a statistician, each row or column represents a *sample* of a population. All these samples have the same size: $n = 10$. Suppose we increase the size to, say, $n = 30$. How will the proportions be distributed then?

Returning to the table, we can see that the first three rows combined contain $6 + 9 + 9 = 24$ ones, the next three rows contain $8 + 7 + 10 = 25$ of them, the first three columns contain $9 + 8 + 9 = 26$, and the next three columns contain $5+10+8 = 23$. Dividing each of these numbers by $n = 30$, we obtain the following proportions: $\frac{24}{30} = 0.80$, $\frac{25}{30} = 0.83$, $\frac{26}{30} = 0.87$, and $\frac{23}{30} = 0.77$. Calculating the average and the standard deviation of these four values, we get 0.82 ± 0.02.

If we compare the results observed in the case of $n = 10$ with those for $n = 30$, we notice two things. First, there is a minor difference between the average in the bigger samples (0.82) and the average in the smaller samples (0.80). Second, the bigger samples exhibit a much smaller standard deviation: 0.02 for $n = 30$ versus 0.08 for $n = 10$. Are these observations mere coincidence, or do they follow from some deeper law?

Estimates Based on Random Samples The answer is provided by a theorem that says that estimates based on samples become more accurate with the growing sample size, n. Further on, the larger the samples, the smaller the variation of the estimates from one sample to another.

Another theorem, the *central limit theorem*, states that the individual estimates follow the Gaussian normal distribution, which we know from Chap. 2—the reader will recall its signature bell-like shape. However, this approximation is reasonably

[1]Recall that *standard deviation* is the square root of *variation*, which is calculated by Eq. 2.13 from Chap. 2.

accurate only if the proportion of ones, p, and the sample size, n, satisfy two fundamental conditions:

$$np \geq 10 \tag{13.1}$$

$$n(1-p) \geq 10 \tag{13.2}$$

If the two conditions are *not* satisfied (if at least one of the products is less than 10), the distribution of estimates obtained from the samples cannot be approximated by the normal distribution without compromised accuracy.

Sections 13.2 and 13.3 will elaborate on how the normal-distribution approximation can help us establish our confidence in the measured performance of the induced classifiers.

Numeric Example Let us check how these conditions are satisfied in the case of the samples in Table 13.1. We know that the proportion of ones in the original population was determined by a user-set parameter of the random-number generator: $p = 0.8$. Let us begin with samples of size $n = 10$. It turns out that none of the two conditions is satisfied because $np = 10 \cdot 0.8 = 8 < 10$ and $n(1-p) = 10 \cdot 0.2 = 2 < 10$. This means that the distribution of the proportions observed in these small samples cannot be approximated by the normal distribution.

In the second attempt, the sample size was increased to $n = 30$. As a result, we obtain $np = 30 \cdot 0.8 = 24 > 10$, which means that Condition 13.1 is satisfied. However, Condition 13.2 is *not* satisfied because $n(1-p) = 30 \cdot 0.2 = 6 < 10$. Even here, therefore, the normal distribution does not offer sufficiently accurate approximation.

The situation will change if we increase the sample size to $n = 60$. Doing the math, we realize that $np = 60 \cdot 0.8 = 48 \geq 10$ and also $n(1-p) = 60 \cdot 0.2 = 12 \geq 10$. We conclude that the distribution of the proportions of ones in samples of size $n = 60$ can be approximated with the normal distribution without perceptible loss in accuracy.

Impact of p Note how the applicability of normal distribution is affected by p, the proportion of ones in the population. It is easy to see that, for different values of p, different sample sizes are called for if the two conditions are to be satisfied. Relatively small size is sufficient if $p = 0.5$; but the more the proportion differs from $p = 0.5$, the bigger the samples we need.

To get a better idea, recall that we found the sample size of $n = 60$ to be sufficient in a situation where $p = 0.8$. What if, however, we decide to base our estimates on samples of the same size, $n = 60$, but in a domain where the proportion is higher, say, $p = 0.95$? In this event, we realize that $n(1-p) = 60 \cdot 0.05 = 3 < 10$, which means that Condition 13.2 is not met, and the proportions in samples of this size cannot be approximated by normal distribution. For this condition to be satisfied in *this* domain, we would need a sample size of at least $n = 200$. Since $200 \cdot 0.05 = 10$, we have just barely made it.

By the way, note that, on account of the symmetry of the two conditions, 13.1 and 13.2, the same minimum size, $n = 200$, will be called for in a domain where $p = 0.05$ instead of $p = 0.95$.

Parameters of the Distribution Let us return to the original task: estimating the proportion of ones based on experience made in samples. We now know that if the samples are large enough, the distribution of these proportions in different samples can be approximated by the normal distribution whose mean equals the (theoretical) proportion of ones in the entire population.

The other parameter of a distribution is the standard deviation. In our context, statisticians prefer the term *standard error*, a terminological subtlety to emphasize the following: whereas "standard deviation" refers to a distribution of *any* variable (such as `weight, age`, or `temperature`), the term "standard error" applies when we refer to variations in estimates from one sample to another. And this is what interests us in the case of our proportions.

Let the standard error be s_E. Mathematicians have established that its value can be calculated from the sample size, n, and the theoretical proportion, p:

$$s_E = \sqrt{\frac{p(1-p)}{n}} \tag{13.3}$$

For instance, if $n = 50$ and $p = 0.80$, then the standard error is as follows:

$$s_E = \sqrt{\frac{0.80 \cdot 0.20}{50}} = 0.06$$

Some engineers prefer to say that the standard error is 6%.

Impact of n: Diminishing Returns Note how the standard error goes the other way than the sample size, n: the larger the samples, the lower the standard error and vice versa. Thus in the case of $n = 50$ and $p = 0.80$, we obtained $s_E = 0.06$. If we use larger samples, say, $n = 100$, the standard error drops to $s_E = \sqrt{\frac{0.8 \cdot 0.2}{100}} = 0.04$. The curve defined by the normal distribution becomes narrower, and the proportions in different samples will be closer to p.

This said, we understand that increasing the sample size brings *diminishing returns*. A simple example will illustrate the point. The calculations from the previous paragraph convinced us that, when proceeding from $n = 50$ to $n = 100$ (doubling the sample size), we managed to reduce s_E by 2% points, from 6 to 4%. If, however, we do the same calculation for $n = 1,000$, we get $s_E = 0.013$, whereas $n = 2000$ results in $s_E = 0.009$. In other words, doubling the sample size from 1000 to 2000 only succeeded in reducing the standard error from 1.3 to 0.9%: the only reward for doubling the sample size was the paltry 0.4%.

This last observation is worth remembering. In many domains, pre-classified examples are difficult or expensive to obtain as, for instance, in the oil-spill domain from Sect. 8.3. If acceptable estimates can be made using a small testing set, the engineer will not want to go into the trouble of procuring additional examples; the puny benefits this may bring will not justify excessive costs.

13.1.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Write down the formulas defining the conditions to be satisfied if the distribution of the proportions obtained from random samples should follow normal distribution.
- Explain how the entire-population proportion affects the sample size necessary for the proportions measured on samples to follow normal distribution.
- Samples have been used to estimate a classifier's error. What do their different sizes tell us about the error? Also, write down the formula that calculates the standard error.
- Elaborate on the statement that "increasing the sample size brings diminishing returns."

13.2 Benefiting from the Normal Distribution

The previous section investigated the proportions of ones in samples. The sample size was n, and the theoretical proportion of ones in the whole population was p. This theoretical value we do not know; we only estimate it based on a sample. Further on, we learned that, while the proportion in each individual sample is different, the distribution of these values can be approximated by the normal distribution—the approximation being reasonably accurate if Conditions 13.1 and 13.2 are satisfied.

Normal distribution can help us decide how much to trust the classification accuracy (or any other performance criterion) measured on a concrete testing set. Let us take a look at how to calculate *confidence values*.

Reformulation in Terms of Classification Performance Suppose the ones and zeros in Table 13.1 represent correct and incorrect classifications, respectively, made by a classifier on a testing set of one hundred examples (one hundred being the number of entries in the table). In this event, the proportion of ones is the classifier's accuracy, whereas the proportion of zeros is its error rate.

Evaluation of the classifier on a different testing set will result in different classification accuracy. But when measured on great many testing sets, the individual accuracies will be distributed in a manner that roughly follows normal distribution.

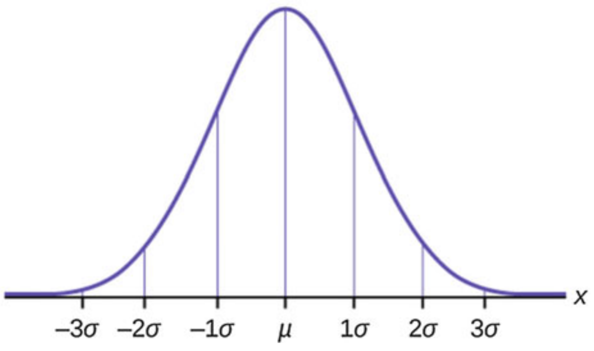


Fig. 13.1 Gaussian (normal) distribution whose mean value is μ

Properties of the Normal Distribution Figure 13.1 shows the fundamental shape of the normal distribution. The vertical axis represents the *probability density function* as we know it from Chap. 2. The horizontal axis represents classification accuracy. The mean value, denoted here as μ, is the *theoretical* classification accuracy which we would obtain when evaluating the classifier on all possible examples from the given domain. This theoretical value is of course unknown; our intention is to estimate it on the basis of a concrete sample, the testing set.

The bell shape of the density function reminds us that most testing sets will yield classification accuracy close to the mean, μ. The greater the distance from μ, the smaller the chance that this particular performance will be measured on a random testing set. Note that the graph highlights certain specific distances from μ along the horizontal axis: the multiples of σ, the distribution's standard deviation—or, if we deal with sample-based estimates, the standard error of these estimates.

The formula defining normal distribution was introduced in Sect. 2.5 as the Gaussian bell function. Knowing the formula, we can establish the percentage of values found within a specific interval, $[a, b]$. The size of the area under the entire curve (from minus infinity to plus infinity) is 1. Therefore, if the area under the curve within the range of $[a, b]$ is 0.80, we can say that 80% of the performance estimates are found in this interval.

Intervals of Interest Not all intervals are equally important. For the needs of classifier evaluation, we are interested only in those that are centered at the mean, μ. Thus the engineer may want to know what percentage of values will be found in the interval $[\mu - \sigma,\ \mu + \sigma]$. Conversely, she may want to know the size of the interval (centered at μ) that contains 95% of all values.

Strictly speaking, questions of this kind are answered with the help of mathematical analysis. Fortunately, we do not need to do the math ourselves; others have done it for us, and we can take advantage of their findings. Some of the most useful results are shown in Table 13.2 where the left column lists percentages called *confidence levels*, and for each of these, the right column specifies the interval that comprises the stated percentage of values. Note that the length of the interval is

Table 13.2 For normal distribution, with mean p and standard deviation σ, the left column gives the percentage of values found in the interval $[\mu - z^*\sigma,\ \ \mu + z^*\sigma]$

Confidence level	Coefficient z^*
68%	1.00
90%	1.65
95%	1.96
98%	2.33
99%	2.58

characterized by z^*, the number of standard deviations to either side of μ. More formally, therefore, the interval is $[\mu - z^*\sigma,\ \ \mu + z^*\sigma]$.

Here is how the table is used. Suppose we want to know the size of the interval that contains 95% of all results. This percentage, 95%, is found in the third row. The right column in this row contains 1.96, which is interpreted as telling us that 95% of all values find themselves in the interval $[\mu - 1.96 \cdot \sigma,\ \ \mu + 1.96 \cdot \sigma]$. Similarly, 68% of the values are in the interval $[\mu - \sigma,\ \ \mu + \sigma]$—this is what follows from the first row in the table.

Standard Error of Sample-Based Estimates How shall we employ these intervals when evaluating classification accuracies? Suppose that the testing sets are all of the same size, n, and suppose that this size satisfies Conditions 13.1 and 13.1 that allow us to assume normal distribution. We already know that the average of the classification accuracies measured on great many independent testing sets will converge to *theoretical accuracy*, the one that would have been obtained by testing the classifier on all possible examples.

The standard error[2] is calculated by Eq. 13.3. For instance, if the theoretical classification accuracy is $p = 0.70$, and the size of each testing set is $n = 100$, then the standard error of the classification accuracies obtained from many different testing sets is calculated as follows:

$$s_{acc} = \sqrt{\frac{p(1-p)}{n}} = \sqrt{\frac{0.7(1-0.7)}{100}} = 0.046 \tag{13.4}$$

After rounding, we say that the classification accuracy is 70% plus or minus 5%. Note, again, that this standard error will be smaller if we use a larger testing set. This makes sense: the larger the testing set, the more thorough the evaluation, and thus the higher our confidence in the obtained result.

Let us now ask what value we are going to obtain if we evaluate the classifier on another testing sets of the same size. Again, we answer the question using Table 13.2. First, we find the row with 95%. In this row, the right column gives coefficient $z^* = 1.96$, which means that 95% of all results will be in the interval

[2]As explained in Sect. 13.1, in our context we prefer the term *standard error* to the more general *standard deviation*.

$[\mu - 1.96 \cdot s_{acc},\ \mu + 1.96 \cdot s_{acc}] = [0.80 - 1.96 \cdot 0.46,\ 0.80 + 1.96 \cdot 0.46] = [0.61, 0.79]$.

Do not forget, however, that this will only be the case if the testing set has the same size, $n = 100$. For a different n, Eq. 13.4 will give us a different standard error, s_{acc}, and thus a different interval.

Reminder Let us remind ourselves why we need the assumption about normal distribution: if the distribution is normal, we can use Table 13.2 from which we learn the size of the interval (centered at μ) that contains the given percentage of values.

On the other hand, the formula for standard error (Eq. 13.3) is valid generally, even if the distribution is *not* normal. For the calculation of standard error, the two conditions, 13.1 and 13.2, do *not* have to be satisfied.

13.2.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- How do the considerations from the previous section apply to the evaluation of an induced classifier's performance?
- What kind of information can we glean from Table 13.2? How can this table be used when quantifying the confidence in the classification accuracy measured on a testing set of size n?
- How will you calculate the standard error of estimates based on a given testing set? How does this standard error depend on the size of the testing set?

13.3 Confidence Intervals

The information from the previous sections helps us quantify the experimenter's confidence in the induced classifier's performance.

Confidence Interval: An Example Now that we understand how the classification accuracies obtained from different testing sets are distributed, we are ready to draw conclusions about how confident we can be in the value measured on one concrete testing set.

Suppose the size of the testing set is $n = 100$, and let the classification accuracy measured on this testing set be $acc = 0.85$. For the training set of this size, the standard error is calculated as follows:

$$s_{acc} = \sqrt{\frac{0.85 \cdot 0.15}{100}} = 0.036 \tag{13.5}$$

Checking the normal-distribution conditions, we realize that they are both satisfied because $100 \cdot 0.85 = 85 \geq 10$ and $100 \cdot 0.15 = 15 \geq 10$. This means that we can take advantage of the z^*-values from Table 13.2. Specifically, we establish that 95% of all values are in the interval $[acc - 1.96 \cdot s_{acc},\ acc + 1.96 \cdot s_{acc}]$. For $acc = 0.85$ and $s_{acc} = 0.036$, the interval is $[0.85 - 0.07,\ 0.85 + 0.07] = [0.78, 0.92]$.

Based on the evaluation on the given testing set, we can therefore conclude that, with 95% confidence, the real classification accuracy finds itself in the *confidence interval* [0.78, 0.92].

Confidence Level and Margin of Error Confidence intervals reflect concrete *confidence levels*—those defined by the percentages listed in the left column of Table 13.2. In the previous paragraph, the confidence level was 95%.

Each confidence level leads to a different confidence interval, which is defined as $\mu \pm M$, where μ is the mean and M is the *margin of error*. For instance, in the case of the interval [0.78, 0.92], the mean was $\mu = 0.85$ and the margin of error was $M = z^* s_{acc} = 1.96 \cdot 0.036 = 0.07$.

Impact of Confidence Level In the previous example, confidence level was 95%, a fairly common choice. If we choose another value, say, 99%, Table 13.2 will give $z^* = 2.8$, which leads to confidence interval $[0.85 - 2.58 \cdot s_{acc},\ 0.85 + 2.58 \cdot s_{acc}] = [0.76, 0.94]$. Note that this interval is broader than the one for 95%. This is logical: the chance that the real classification accuracy finds itself in a longer interval is greater. Conversely, it is less likely that the theoretical value will fall into a narrower interval. Thus for the confidence level of 68% (and the standard error rounded to $s_{acc} = 0.04$), the confidence interval is $[0.85 - 0.04,\ 0.85 + 0.04] = [0.81;\ 0.89]$.

Let us not forget that, even in the case of confidence level of 99%, one cannot be absolutely sure that the theoretical value will fall into the corresponding interval. There is still that 1% probability that the measured value will be outside this interval.

Another Parameter: Sample Size The reader now understands that the length of the confidence interval depends on the standard error and that the standard error, in turn, depends on the size, n, of the testing set (Eq. 13.3). Essentially, the larger the testing set, the stronger the evidence in favor of the measured value, and thus the narrower the confidence interval. We say that the margin of error and the training-set size are in *inverse relation*: as the size of the training set increases, the margin of error decreases.

Earlier, we mentioned that a higher confidence level results in a longer confidence interval. If we think this interval is too big, we can make it shorter by using a bigger testing set and thus a higher value of n (which decreases the standard error).

There is a way to decide how large the testing set should be if we want to limit the margin of error to a certain maximum. Here is the formula for calculating the margin of error:

$$M = z^* s_{acc} = z^* \sqrt{\frac{p(1-p)}{n}} \tag{13.6}$$

Solving this equation for n (for specific values of M, p, and z^*) will give us the required testing-set size.

Concluding Remark The method of establishing the confidence interval for a given confidence level was explained using the simplest performance criterion, classification accuracy. Yet the scope of the method's applicability is much broader: the uncertainty of *any* variable that represents a proportion can thus be quantified. In the context of machine learning, we can use the same approach to establish our confidence in any of the performance criteria from Chap. 12, be it *precision*, *recall*, or some other quantity.

However, we have to do it right. For one thing, we must not forget that the distribution of the values can only be approximated by normal distribution if Conditions 13.1 and 13.2 are satisfied. Second, we have to understand the meaning of n when calculating the standard error by Eq. 13.3. For instance, the reader remembers that *precision* is calculated with the formula, $\frac{N_{TP}}{N_{TP}+N_{FP}}$: the percentage of true positives among all examples labeled by the classifier as positive. This means that we are dealing with a proportion of true positives in a sample whose size is $n = N_{TP} + N_{FP}$. Similar considerations apply in the case of *recall*, *sensitivity*, and other performance criteria.

13.3.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Explain the meaning of *confidence interval*. What is *margin of error*?
- How does the size of the confidence interval (and the margin of error) depend on the user-specified value of confidence level? How does it depend on the size of the testing set?
- Discuss the calculations of confidence intervals for some other performance criteria such as *precision* and *recall*.

13.4 Statistical Evaluation of a Classifier

A claim about a classifier's performance can be confirmed or refuted experimentally, by testing the classifier on a set of pre-classified examples. One way to statistically evaluate the results is by the algorithm in Table 13.3. Let us illustrate the procedure on a simple example.

Numeric Example Suppose a machine-learning specialist tells you that a concrete classifier has classification accuracy $acc = 0.78$. Faithful to the dictum, "trust but verify," you decide to find out whether this statement is correct. To this end,

Table 13.3 The algorithm for statistical evaluation of a classifier's performance

(1) For the given size, n, of the testing set and for the claimed classification accuracy, acc, check whether the conditions for normal distribution are satisfied:
$n \cdot acc \geq 10$ and $n \cdot (1 - acc) \geq 10$
(2) Calculate the standard error by the following formula:
$s_{acc} = \sqrt{\frac{acc(1-acc)}{n}}$
(3) Assuming that the normal-distribution assumption is correct, find in Table 13.2 the z^*-value for the requested level of confidence. The corresponding confidence interval is $[acc - z^* \cdot s_{acc},\ acc + z^* \cdot s_{acc}]$.
(4) If the value measured on the testing set falls outside this interval, reject the claim that the accuracy is acc. If the value falls inside the interval, the evidence for rejection is insufficient.

you prepare $n = 100$ examples whose class labels are known and then set about measuring the classifier's performance on this testing set.

Let the experiment result in a classification accuracy of 0.75. This is less than the promised 0.78, but then: is the difference within reasonable bounds? Is there a chance that the specialist's claim was correct and that the lower performance measured on the testing set can be explained by the variations due to the random nature of the testing data? After all, a different testing set is likely to result in different classification accuracy!

Checking the Conditions for Normal Distribution The first question to ask is whether the distribution of the performances thus obtained can be approximated by normal distribution. A positive answer will allow us to base our statistical evaluation on the values in Table 13.2.

Conditions 13.1 and 13.2 are easily verified. Seeing that $np = 100 \cdot 0.75 = 75 \geq 10$ and that $n(1 - p) = 100 \cdot 0.25) = 25 \geq 10$, we realize that the conditions are satisfied, and the normal-distribution assumption is justified.

Confidence Interval for the 95% Confidence Level Suppose you are prepared to accept the specialist's claim ($acc = 0.78$) if there is at least 95% chance that the observed classification accuracy (on a random testing set) is $acc = 0.75$. Does the value, 0.75, find itself within the confidence interval that is centered at 0.78? Let us find out.

The corresponding row in the table informs us that $z^* = 1.96$. This means that 95% of the results obtained on random testing sets will fall in the interval $[acc - 1.96 \cdot s_{acc},\ \ acc + 1.96 \cdot s_{acc}]$, where $acc = 0.78$ is the original claim, and s_{acc} is the standard error to be statistically expected for testing sets of size n.

The size of our testing set is $n = 100$. The standard error is calculated as follows:

$$s_{acc} = \sqrt{\frac{acc(1 - acc)}{n}} = \sqrt{\frac{0.75 \cdot 0.25}{100}} = 0.043 \tag{13.7}$$

We conclude that the confidence interval is $[0.78 - 1.96 \cdot 0.043,\ \ 0.78 + 1.96 \cdot 0.043]$, which, after evaluation and rounding, is $[0.70, 0.86]$.

Conclusion Regarding the Specialist's Claim Evaluation on our own testing set resulted in classification accuracy $acc = 0.75$, a value that finds itself within the confidence interval corresponding to confidence level of 95%.

This is encouraging. For the claim, $acc = 0.78$, there is a 95% probability that evaluation on a random testing set will result in classification accuracy within interval $[0.70, 0.86]$. This is what happened in this particular case. Although our result, $acc = 0.75$, is somewhat lower than the specialist's claim, we have to admit that our experimental evaluation failed to provide convincing evidence to refute the claim. In the absence of this evidence, we accept the claim as valid.

Type-I Error in Statistical Evaluation: False Alarm Here is the principle. Someone makes a claim about performance. Based on the size of our testing set (and assuming normal distribution), we calculate the size of the interval that is supposed to contain the given percentage of experimental results. If, for instance, the percentage is 95%, and if the claim is correct, then there is a 5% chance that a testing result will fall outside this interval. We reject the original claim if the testing result landed in the less-than-5% region outside the interval, believing that it is unlikely that such difference would be observed if the claim were correct.

This said, we must not forget that there is a 5% probability that such difference will be observed: there is a danger that the classifiers' evaluation on the testing set will result in a value outside the given confidence interval. In this case, rejecting the specialist's claim is unfair. Statisticians call this the *type-I* error: the false rejection of an otherwise correct claim, a rejection due to non-typical results.

If we are concerned about the risk of committing type-I error, we can reduce it by increasing the required confidence level. If we choose 99% instead of the 95%, false alarms will be less frequent. But this reduction does not come for free—as will be explained in the next paragraph.

Type-II Error in Statistical Evaluation: Failing to Detect Incorrect Claim Also the opposite case is possible; to wit, the initial claim is false, and yet the value measured on the testing set falls inside the confidence interval. The experiment failed to provide sufficient evidence against the claim which thus has to be accepted.

This is what sometimes happens: a false claim fails to be refuted. Statisticians call this the *type-II* error. It is typical of applications that call for high confidence levels: so wide is the resulting confidence interval that the results of testing will almost never land outside it, which means that an experiment rarely leads to the claim's rejection.

The thing to remember is the trade-off between the two types of error. By increasing the confidence level, we reduce the risk of the *type-I* error, but only at the cost of increasing the danger of the *type-II* error and vice versa.

13.4.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Explain the principles of the statistical approach to evaluating the engineer's confidence in experimental results.
- What is meant by *type-I error* (false alarm)? What can be done to reduce the danger of making this error?
- What is meant by *type-II error* (fail to detect)? What can be done to reduce the danger of making this error?

13.5 Another Use of Statistical Evaluation

The reader now understands the principles of statistical processing of experimental results and knows how to employ them when evaluating the performance of a classifier. However, statistics can do much more.

Do Two Testing Sets Represent Different Contexts? Chapter 11 explained that sometimes a different classifier should perhaps be induced for a different context—such as British accent versus American accent. Here is how statistics can help us identify such situations in the data.

Suppose we have tested a classifier on two different testing sets. The classification accuracy in the first test is $\hat{p}_1$ and the classification accuracy in the second test is $\hat{p}_2$ (the letter "p" alluding to proportion of correct answers). The sizes of the two sets are denoted by n_1 and n_2. Finally, let the average proportion of correctly classified examples in the two sets combined be denoted by $\hat{p}$.

The statistics of interest is defined by the following formula:

$$z = \frac{\hat{p}_1 - \hat{p}_2}{\sqrt{\hat{p}(1-\hat{p})(\frac{1}{n_1} + \frac{1}{n_2})}} \tag{13.8}$$

The result is compared to the critical value for the given confidence level—the value found in Table 13.2.

Numeric Example Suppose the classifier was evaluated on two testing sets whose sizes are $n_1 = 100$ and $n_2 = 200$. Let the classification accuracies measured on the two sets be $\hat{p}_1 = 0.82$ and $\hat{p}_2 = 0.74$, respectively, so that the average classification accuracy on the two sets combined is $\hat{p} = 0.77$. The reader will easily verify that the conditions for the use of normal distribution are satisfied.

Plugging these values into Eq. 13.8, we obtain the following:

$$z = \frac{0.82 - 0.74}{\sqrt{0.77(1-0.77)(\frac{1}{100}+\frac{1}{200})}} = 1.6. \tag{13.9}$$

Since this value is lower than the one given for the 95% confidence level in Table 13.2, we conclude that the result is within the given confidence interval and therefore accept that the two results are statistically indistinguishable.

13.5.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Why should we be concerned that a classifier is being applied to a wrong kind of data?
- What is the formula used here? How do you carry out the evaluation?

13.6 Comparing Machine-Learning Techniques

Sometimes we need to decide which of two alternative machine-learning techniques is more appropriate for the problem at hand. The methodology relies on principles similar to those adopted in the previous sections, and yet there is a small difference. Let us illustrate the approach on simple example.

Experimental Methodology As discussed in Chap. 12, one way to compare, experimentally, two machine-learning algorithms is to rely on 5x2 cross-validation. In this method, the set of available pre-classified data is divided into two equally sized subsets, T_{11} and T_{12}. First, the two machine-learning techniques induce their classifiers from T_{11}, and the induced classifiers are then tested on T_{12}; in the next step, the two classifiers are induced from T_{12} and tested on T_{11}. The process is repeated five times, each time for a different random division of the set of data into two subsets, T_{i1} and T_{i2}.

This results in ten pairs of testing-set classification accuracies (or error rates, precisions, recalls, or any other performance criterion of choice). Here is the question: "Are the differences between the ten pairs of results statistically significant?"

Experimental Results: Paired Comparisons Let us denote the i-th pair of sets by T_{i1} and T_{i2}, respectively. Suppose we are comparing two machine-learning algorithms, ML1 and ML2, evaluating them in the ten experimental runs from the

Table 13.4 Example experimental results of a comparison of two alternative machine-learning techniques, ML1 and ML2. The numbers in the first two rows give classification accuracies (percentages), and the last row gives the differences, d

	T_{11}	T_{12}	T_{21}	T_{22}	T_{31}	T_{32}	T_{41}	T_{42}	T_{51}	T_{52}
ML1	78	82	99	85	80	95	87	57	69	73
ML2	72	79	95	80	80	88	90	50	73	78
d	6	3	4	5	0	7	-3	7	-4	-5

previous paragraph, and suppose the results are those listed in Table 13.4. Here, each column is headed with the name of the test set used in the given experimental run. The fields in the table give the classification accuracies achieved on the test sets by classifiers induced by the two alternative techniques. The last row specifies the differences between the two classification accuracies. Note that the differences can be positive as well as negative.

Evaluating these results, we realize that the average difference is $\overline{d} = 2.0$ and that the standard deviation of these differences is $s_d = 4.63$.

Statistical Evaluation of Paired Differences Observing the mean value of differences, $\overline{d}$ (with standard deviation, s_d), we have to ask: is this difference statistically significant? In other words, is this difference outside what we previously called a confidence interval for the given confidence level, say, 95%? Note that the midpoint of this confidence interval is $\hat{d} = 0$.

Let us point out two major differences between what we need here and what we needed in the previous sections. First, we now work with mean values, d, instead of proportions. Second, we can no longer rely on normal distribution because the number of experiments is small, and the standard deviation has only been estimated on the basis of the given 10 observations (the standard deviation is not known for the entire population).

In this case, we resort to another statistical model, the t-distribution, which is similar to normal distribution in its bell shape but is flatter. Its "flatness" or "steepness" depends on what is called the number of *degrees of freedom*. In the case of 10 testing sets, there are $10 - 1 = 9$ degrees of freedom. Some typical t values for nine degrees of freedom are shown in Table 13.5.[3]

[3]With more degrees of freedom, the curve would get closer to normal distribution, becoming almost indistinguishable from it for 30 or more degrees of freedom.

Table 13.5 Some probabilities of the t-values for nine degrees of freedom

	Confidence level			
Degrees of freedom	0.10%	0.05%	0.02%	0.01%
9	1.83	2.26	2.81	3.35

Calculating t-Values in Paired Tests Statistical evaluation with t-tests is essentially the same as in the case of normal distribution. For the mean difference, $\overline{d}$, and the standard deviation, s_d, the t_9-value (the subscript specifies the number of degrees of freedom) is calculated by the following formula, where n is the number of tests:

$$t_9 = \frac{\overline{d} - 0}{s_{\overline{d}}/\sqrt{n}} \tag{13.10}$$

The result is then compared with the thresholds associated with concrete levels of confidence listed in Table 13.5. Specifically, in the case of the results from Table 13.4, we obtain the following:

$$t_9 = \frac{2 - 0}{4.63/\sqrt{10}} = 1.35 \tag{13.11}$$

Seeing that this is less than the 2.26 listed in the table for the 95% confidence level, we conclude that the experiment has failed to refute (for the chosen confidence level) the hypothesis that the two techniques lead to comparable classification accuracies. We therefore accept the claim.

Word of Caution One thing is easily overlooked. It is not enough that one machine-learning software leads to better results than another, and it not sufficient to show this improvement to be statistically significant. *Significantly better* (statistically) is not the same as *much better*. 0.5% improvement in classification performance, even if trusted with 95% confidence, may be meaningless if the minuscule improvement can only be obtained with exorbitant programming and experimentation costs and if it presents difficulties with scaling to larger or different data. Beyond the realm of statistics, the conscientious engineer reflects the practical needs of the material world.

13.6.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Explain the principle of the 5x2 cross-validation Technique, which leads to the set of 10 paired results.

- Why cannot we use normal distribution as we did in the previous sections? What other distribution is used here?
- Write down the formula for calculating the t-value. Explain how the value for each individual variable in this formula is obtained.

13.7 Summary and Historical Remarks

- The essence of statistical evaluation is to draw conclusions about the behavior of an entire population from observations made on a relatively small sample.
- Different samples will yield different results; however, the distribution of these results has to satisfy the laws of statistics. Knowledge of these laws helps the engineer to calculate the confidence in the classification performance obtained on a concrete testing set.
- The most typical distribution of "sampling results" is the Gaussian normal distribution. It can be used only if two conditions are satisfied. For training-set size n and the mean value p, the conditions are $np \geq 10$ and $n(1-p) \geq 10$.
- Regardless of the distribution being normal or not, the standard error of the accuracies, acc, measured on different testing sets is calculated as follows:

$$s_{acc} = \sqrt{\frac{acc(1-acc)}{n}} = \sqrt{\frac{0.75 \cdot 0.25}{100}} = 0.043$$

- For each confidence level, the normal-distribution assumption leads to a specific z^* value (see Table 13.2), Having calculated the standard error and having chosen a confidence level, we establish the confidence interval by the following formula:

$$[acc - z^* s_{acc},\ acc + z^* s_{acc}]$$

  The term $z^* s_{acc}$ is called *margin of error*. For other performance metrics (e.g., *sensitivity*, similar formulas are used.
- Suppose we are testing the claim that classification accuracy is acc. If the experimental result falls into the confidence interval for the chosen confidence level, we do not have enough evidence against the claim regarding the value of acc. If the result is outside the value, the evidence is sufficient and we reject the claim.
- When comparing two machine-learning techniques, we often employ 5x2 cross-validation with t-test that uses t-distribution instead of normal distribution. The t-distribution has a slightly different shape for different numbers of *degrees of freedom*.

Historical Remarks The statistical methods discussed in this chapter are so old and well established that textbooks of statistics no longer care to give credit to those who developed them. From the perspective of machine learning, however, it is important to note that the idea of applying t-tests to experimental results obtained from 5x2 cross-validation was advocated by Dietterich (1998).

13.8 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

13.8.1 Exercises

1. Suppose we want to evaluate the claim that a classifier's accuracy is $p = 0.80$. What size, n, of the testing set will allow us to rely on normal distribution?
2. Suppose a classifier's accuracy has been specified as $p = 0.9$. What is the value of the standard error, s_E, if the classifier is evaluated on a testing set of size $n = 400$? Determine the size of the confidence interval for the 95% confidence level. Do not forget to check the validity of Conditions 13.1 and 13.2.
3. Your company considers a classifier with stated accuracy $p = 0.85$. This accuracy was evaluated on a testing set of 200 examples, the measured value being 0.81, which of course is less than what was promised. Is there at least 95% chance that the original claim was correct? What about 99%?

13.8.2 Give it Some Thought

1. Suppose you test a classifier's performance, using the 95% confidence interval. What if you change your mind and decide to use the 99% confidence instead? You will increase tolerance, but what is the price for this?
2. Write an essay summarizing the trade-offs between narrower confidence intervals on the one hand and the need to purchase possibly expensive testing examples.

13.8.3 Computer Assignments

1. Write a program that receives the following input: stated performance, performance measured on a testing set, the size of this testing set, and the user-specified level of confidence. The program outputs a Boolean variable that is *true* if the stated value can be accepted and *false* if it cannot be accepted.

2. This assignment assumes that the reader has already implemented a program dividing the data into the five "folds" needed for the 5x2 CV methodology. Another assumption is that the reader has implemented at least two class-induction programs.
   Write a program that compares the two induction techniques using the 5x2 CV methodology, evaluating the results using t-tests.

Chapter 14
Induction in Multi-label Domains

Up till now, we have always assumed that each example is labeled with one and only one class. In realistic applications, however, this is not always the case. Quite often, an example is known to belong to two or more classes at the same time, sometimes to *many* classes. This situation presents certain new problems whose nature the engineer needs to understand.

After a brief discussion of how to handle multi-label domains within the framework of classical paradigms, this chapter focuses on the currently most popular approach: *binary relevance*. The idea is to induce a binary classifier separately for each class and then to use all these classifiers in parallel. More advanced versions of this technique seek to improve classification performance by exploiting mutual interrelations between classes, and yet another alternative employs the mechanism of *class aggregation*. Attention is paid also to the specific aspects of performance evaluation in multi-label domains.

14.1 Classical Paradigms and Multi-label Data

Let us begin with an informal definition of the multi-label domain and then take a look at how to address such data within the classical paradigms that the reader has already mastered.

What Is a Multi-label Domain? In many applications, the traditional requirement that an example be labeled with one and only one class is hard to satisfy. Thus a text document may represent `nutrition`, `diet`, `athletics`, `popular science`, and perhaps quite a few other categories. Alternatively, a visual image may at the same time represent `summer`, `cloudy weather`, `beach`, `sea`, `seagulls`, and so on. Something similar is observed in many other domains.

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_14

The number of classes of an average example differs from one application to another. In some, almost every example has a great many labels selected from perhaps thousands of different classes. At the other extreme are domains where only some examples belong to more than one class, the majority having only one label.

Whatever the characteristics of concrete data, the goal is to induce a classifier (or a set of classifiers) satisfying two basic requirements. First, the tool should for a given example return as many of its true classes as possible; missing any one of them constitutes a *false negative*. Second, the classifier should not label the example with a class to which the example does not belong; each such wrong class constitutes a *false positive*.

Neural Networks Chapter 6 explained the essence of *multilayer perceptrons* (MLP), a popular architecture of artificial neural networks. The reader will recall that the output layer consists of one neuron for each class, the number of inputs being equal to the number of attributes. The ideal size of the hidden layer then reflects the complexity of the classification task at hand.

On the face of it, using an MLP in multi-label domains should not pose any major problems. For instance, suppose the network has been presented with a training example that is labeled with classes C_3, C_6, and C_7. In this event, the target values for training will be set to, say, $t_i = 0.8$, in the case of output neurons with indices $i \in \{3, 6, 7\}$ and to $t_i = 0.2$ for all the other output neurons.[1] The backpropagation-of-error technique can then be used in the same way as in single-label domains.

Word of Caution Multilayer perceptrons may not necessarily be the best choice here. Indeed, neural-network literature has devoted to multi-label domains much less attention than to other tasks, and not without reason. For one thing, the training of plain MLPs is known to be vulnerable to local minima, and there is always the architecture-related conundrum: how many hidden neurons will strike a sensible compromise between overfitting the data if the network is too large and lacking sufficient flexibility if the network is too small?

Also the notoriously high computational costs are a reason for concern. The fact that each training example can belong to more than one class complicates things. Sensing the difficulty, the engineer is tempted to increase the number of hidden neurons—which not only adds to the already high computational costs but also increases the danger of overfitting.

Particularly unclear is the question how exactly to go about classifications. Suppose the forward-propagation step in a six-output MLP resulted in the following output vector: $(0.7, 0.3, 0.8, 0, 5, 0.4, 0.2)$. Does this mean that the example belongs to classes C_1 and C_3? What about C_3? Where exactly should we draw the line?

Do not forget that training neural networks is more art than science. While a lot can be achieved with ingenuity and experience, beginners are often disappointed.

[1]The reader will recall that the target values 0.8 and 0.2 are more appropriate for backpropagation-of-error with `sigmoids` than 1 and 0; see Chap. 6.

In the case of failure, the engineer has to resort to some alternative less dangerous technique.

Nearest-Neighbor Classifiers Another possibility is the nearest-neighbor classifier from Chap. 3. When example **x** is presented, the k-NN classifier first identifies the example's k nearest neighbors. Each of these may be labeled with a set of classes, and the simplest classification attempt will label **x** with the union of these sets. For instance, suppose that $k = 3$, and let that the sets of class labels encountered in the three nearest neighbors be $\{C_1, C_2\}$, $\{C_2\}$, and $\{C_1, C_3\}$, respectively. In this event, the classifier will classify **x** as belonging to C_1, C_2, and C_3.

Word of Caution This approach is practical only in domains where the average number of classes per example is moderate, say, less than three. Also the number of voting neighbors, k, should be small. If these two requirements are violated, too many class labels may be returned for **x**, and this can cause many false positives, which means poor *precision* and *specificity*. At the same time, however, the multitude of returned labels tends to reduce the number of false negatives, which improves *recall* and *sensitivity*. In some domains, this is what we want; in others, *precision* and *specificity* are critical. The users need to know what they want.

As so often in this paradigm, the engineer must resist the temptation to increase the number of the nearest neighbors in the hope that spreading the vote over more participants will give a chance to less frequent classes. The thing is, some of these "nearest neighbors" may be too distant from **x** and thus inappropriate for classification.

Other Approaches Machine-learning scientists have developed quite a few other ways of modifying traditional machine-learning paradigms to make them applicable to multi-label domains. Among these, very interesting are attempts to induce multi-label decision trees. But since they are somewhat too advanced for an introductory text, we will not present them here. After all, comparable classification performance can be achieved by simpler means. These will be the subject of the rest of this chapter.

14.1.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Suggest an example of a multi-label domain. What is the essence of the underlying machine-learning task? In what way will one multi-label domain differ from another?
- Explain the simple method of multi-label training in multilayer perceptrons. What practical difficulties will discourage you from using this paradigm?
- Describe the simple way of addressing a multi-label domain by a k-NN classifier. Discuss its potential pitfalls.

14.2 Principle of Binary Relevance

Let us now proceed to the currently most popular approach, the technique of *binary relevance*. We will begin by explaining the principle and then identify some of shortcomings and limitations that will then be addressed by the following sections.

Binary Relevance The most common approach to multi-label domains induces a separate binary classifier for each class: in a domain with N classes, N classifiers are induced. When classifying a future example, all these classifiers are used in parallel, and the example receives all classes whose classifiers returned the positive label.

For the induction of these classifiers, the training data have to be modified accordingly. Here is how. For the i-th class ($i \in [1, N]$), we create a training set, T_i, that consists of the same examples as the original training set, T, the only difference being in labeling: in T_i, an example's class label is 1 if the list of class labels for this example in T contains C_i; otherwise, the label in T_i is 0.

Once the new training sets have been created, we apply to each of them a *baseline learner* that is responsible for the induction of the individual classifiers. Common practice applies the same baseline learner to each T_i. Typically, we use to this end some of the previously discussed machine-learning techniques such as perceptron learning, decision tree induction, and so on.

Illustration Table 14.1 illustrates the mechanism that creates the new training data. In the original training set, T, five different class labels can be found: $C_1, \ldots, C_5$. The *binary relevance* technique creates the five new training sets, $T_1, \ldots, T_5$, shown in the five tables below the original one. Thus in the very first, T_1, examples ex$_1$

Table 14.1 The original multi-label training set is converted into five new training sets, one for each class

	Classes
ex$_1$	C_1, C_2
ex$_2$	C_2
ex$_3$	C_1, C_3, C_5
ex$_4$	C_2, C_3
ex$_5$	C_2, C_4

T_1	
ex$_1$	1
ex$_2$	0
ex$_3$	1
ex$_4$	0
ex$_5$	0

T_1	
ex$_1$	1
ex$_2$	1
ex$_3$	0
ex$_4$	1
ex$_5$	1

T_1	
ex$_1$	0
ex$_2$	0
ex$_3$	1
ex$_4$	1
ex$_5$	0

T_1	
ex$_1$	0
ex$_2$	0
ex$_3$	0
ex$_4$	0
ex$_5$	1

T_1	
ex$_1$	0
ex$_2$	0
ex$_3$	1
ex$_4$	0
ex$_5$	0

and $\mathtt{ex}_3$ are labeled with 1 because these (and only these) two examples contain the label C_1 in the original T. The remaining examples are labeled with 0.

The baseline learner is applied separately to each of the five new sets, inducing from each T_i the corresponding classifier C_i.

Easy-to-Overlook Detail In each of the training sets thus obtained, every example is labeled as a positive or negative representative of the given class. When the induced binary classifiers are used in parallel (to classify some $\mathbf{x}$), it may happen that none of them returns 1. This means that no label for $\mathbf{x}$ has been identified. When writing the machine-learning software, we must not forget to instruct the classifier what to do in this event. Usually, the programmer chooses from the following two alternatives: (1) return a default class, perhaps the one most frequently encountered in T or (2) reject the example as too ambiguous to be classified.

Discussion The thing to remember is that *binary relevance* transforms the multi-label problem into a set of single-label tasks that are then addressed by classical techniques. To avoid disappointment, however, the engineer needs to be aware of certain difficulties which, unless properly addressed, may compromise performance. Let us briefly address them.

Problem 1: Imbalanced Classes Some of the new training sets, T_i, are likely to suffer from the problem of *imbalanced class* representation , which was discussed in Sect. 11.2. In Table 14.1, this happens to sets T_4 and T_5. In each, only one example out of five (20%) is labeled as positive, all others being negative. We already know that, in situations of this kind, machine-learning techniques are biased toward the majority class—in this case, the class labeled as 0.

The most straightforward approaches to address this issue are classifier modification, majority-class undersampling, and minority-class oversampling. Which of them to choose will depend on the domain's circumstances. As a rule of thumb, one can base the decision on the size of the training set. In very big domains, majority-class undersampling is preferred. When the examples are scarce, the engineer cannot afford to "squander" them and prefers minority-class oversampling. Classifier modification is somewhere between the two.

Problem 2: Computational Costs Some multi-label domains are very large. Thus the training set in a text categorization domain may consist of hundreds of thousands of examples, each described by tens of thousands of attributes and labeled with a subset of thousands of different classes. It stands to reason that to induce thousands of decision trees from a training set of this size will be expensive, perhaps prohibitively so. This means that, when considering candidates for the baseline learner, we have to pay attention to computational costs.

Another possibility is the technique that Sect. 9.5 discussed in the context of *boosting*: for each class, create multiple subsets of the training examples, some of them perhaps described by different subsets of attributes. The idea is to induce for each class a group of sub-classifiers that will vote. If learning from 50% of the examples takes only 5% of the time, considerable savings can be achieved.

Problem 3: Performance Evaluation Another question is how to measure the success or failure of the induced group of classifiers. Usually, each of them will exhibit different performance, some better than average, others worse, and yet others dismal. To get an idea of the big picture, some averaging of the results is needed. We will return to this issue in Sect. 14.7.

Problem 4: Mutual Interdependence of Classes The basic version of *binary relevance* treats all classes as if they were independent of one another. Quite often, this assumption is justified. In other domains, the classes *are* interdependent, but not much harm is done by ignoring this fact. In some applications, however, the overall performance of the induced classifiers considerably improves if we find a way to exploit the class interdependence. Techniques for doing so will be discussed in the next three sections.

14.2.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Describe the principle of *binary relevance*. How does it organize the learning process, and how are the induced classifiers used for the classification?
- What aspect can render the computational costs of *binary relevance* prohibitively high? What can the engineer do to make them acceptable?
- Why does *binary relevance* often lead to the problem of imbalanced classes? What remedies would you recommend?

14.3 Classifier Chains

The information that a text document has been labeled as `nutrition` is sure to increase the chances of its belonging also to `diet`—and decrease the likelihood that it has anything to do with `quantum mechanics`. In the context of binary relevance, this means that exploiting class interdependence may improve classification performance. One possibility is known as *classifier chain*.

Classifier Chain The idea is to induce a chain of classifiers such as the one in Fig. 14.1. The leftmost classifier is induced from the original examples that are labeled as positive or negative instances of class C_1 (recall the training set T_1 from Table 14.1). The second classifier is then induced from examples labeled as positive or negative instances of class C_2. To describe these latter examples, however, one extra attribute is added to the original attribute vector: the output of C_1. The same

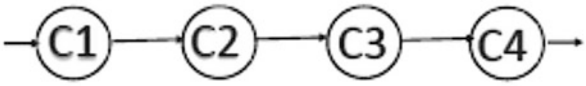


Fig. 14.1 With the exception of $C1$, the input of each classifier consists of the original attribute vector *plus* the labels returned by the previous classifiers

principle is then repeated in the induction of the remaining classifiers: for each, the training examples are described by the original attribute vector *plus* the class label returned by the previous classifier.

When using the classifier chain for the classification of a future example **x**, the same pattern is followed. The leftmost classifier receives **x** that is described by the original attributes. To all other classifiers, the system presents **x** that is described by the original attribute vector plus the labels delivered by the previous classifiers. Ultimately, **x** is labeled with those classes whose classifiers returned 1.

Important Assumption (Rarely Satisfied) In the classifier-chain technique, the ordering of the classes from left to right is the engineer's responsibility. In some applications, this is easily done because the classes form a logical sequence. Thus in document classification, `science` subsumes `physics`, which in turn subsumes `quantum mechanics`, and so on. If a document does not belong to `science`, it is unlikely to belong to `physics`, either; it thus makes sense to choose `science` as the leftmost node in the graph in Fig. 14.1 and to place `physics` next to it.

In other applications, class subsumption is not so obvious, but the sequence can still be used without impairing the overall performance. Even when the subsumptions are only intuitive, the engineer may always resort to a sequence backed by experiments: he or she can suggest a few alternative versions, test them, and then choose the one which offers the best results. Another possibility is to create the classifier chain only for *some* of the classes (where the interrelations are known) and apply to the others plain *binary relevance*.

Hierarchically Ordered Classes Class interrelation does not have to be linear. It can acquire forms that can only be reflected by a more sophisticated data structure, perhaps a graph. In that case, we will need more advanced techniques such as the one described in Sect. 14.5.

One Shortcoming of the Classifier-Chain Approach More often than not, the engineer lacks any a priori knowledge about class interrelations. If classifier chains are still to be employed, the best that the engineer can do is to create the classifier sequence randomly. Of course, such *ad hoc* method cannot be guaranteed to work; a poorly designed classifier sequence may be so harmful that the performance of the induced system may be well below that of plain *binary relevance*.

Sometimes, however, there is a way out. If the number of classes is manageable (say, five), the engineer may choose to experiment with several alternative sequences and then choose the best. If the number of classes is greater, though, the attempt to try many alternatives will be impractical.

Error Propagation The fact that the classifiers are forced into a linear sequence makes them vulnerable to *error propagation*. Here is what it means. When a classifier misclassifies an example, the incorrect class label is passed on to the next classifier that uses this label as an additional attribute. An incorrect value of this additional attribute may then sway the next classifier to a wrong decision, which, too, is then passed on down the chain. In other words, an error of a single class may cause errors of subsequent classifiers. In this event, the *classifier chain* will underperform. The thing to remember is that the overall error rate depends on the quality of the classifiers higher-up in the sequence.

One Last Comment The error-propagation phenomenon is less damaging if the classifiers do not strictly return either 0 or 1. Thus a Bayesian classifier calculates for each class its probability, a number from the interval [0, 1]. Propagating this probability through the chain is less harmful than propagating strictly 1 or 0. Similar considerations apply to some other classifiers such as neural networks: multilayer perceptrons and radial basis function networks.

14.3.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Discuss the impact of class interdependence on the performance of *binary relevance*.
- Explain the principle of *classifier chain*. What can you say about the need to find a proper sequence of classifiers?
- Explain the problem of *error propagation* in classifier chains. Is there anything else to criticize in this approach?

14.4 Another Possibility: Stacking

In the light of the weaknesses of *classifier chains*, some alternatives are needed. One of them is *stacking*.

Architecture and Induction in Stacking The essence is illustrated in Fig. 14.2. The classifiers are arranged in two layers. The upper one represents plain *binary relevance*—an independently induced binary classifier for each class. More interesting is the bottom layer. Here, the classifiers are induced from the training sets where the original attribute vectors are extended by the list of the class labels returned by the upper-layer classifiers. In the concrete case depicted in Fig. 14.2,

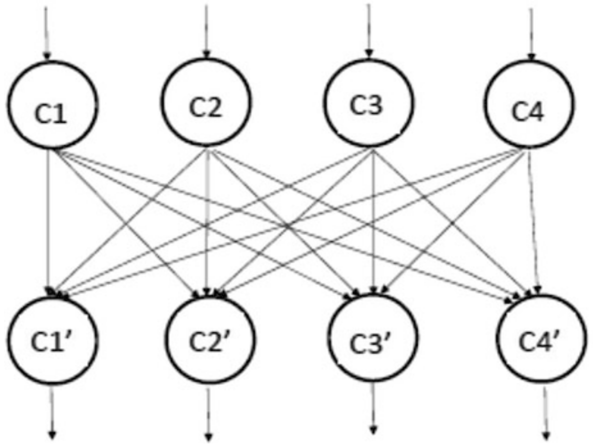


Fig. 14.2 *Stacking*. The upper-layer classifiers use as input the original attribute vector. For the lower-layer classifiers, this vector is extended by the class labels returned by the upper layer

each attribute vector is preceded by four additional binary attributes (because there are four classes): the i-th of these new attributes has value 1 if the i-th classifier in the upper layer has labeled the example with class i; otherwise, this attribute's value is 0.

Classification When the class labels of some future example **x** are needed, **x** is presented first to the upper-layer classifiers. In the next step, the obtained class labels are added at the front of **x**'s original attribute vector as N new binary attributes (if there are N classes), and the newly described example is presented in parallel to all lower-layer classifiers. Finally, **x** is labeled with the classes whose lower-layer classifiers have returned 1.

The underlying philosophy rests on the intuition that the performance of classifier C_i may improve if this classifier is informed about the "opinions" of the other classifiers—about the other classes to which **x** belongs.

Numeric Example Consider an example described by a vector of four attributes with the following values: $\mathbf{x} = \{a, f, r, z\}$. Suppose the upper-layer classifiers return the following labels: $C_1 = 1, C_2 = 0, C_3 = 1, c_4 = 0$. In this event, the lower-layer classifiers are all presented with the following example description: $\mathbf{x} = \{1, 0, 1, 0, a, f, r, z\}$.

The classification decisions of the lower-level classifiers can differ from those in the upper layer. For example, if the lower-layer classifiers return 1, 1, 1, 0, the overall system will label **x** with C_1, C_2, and C_3, ignoring the original recommendations of the upper layer.

Comments This approach is more flexible than *classifier chains* because *stacking* makes it possible for any class to affect the identification of the other classes. The engineer does not provide any a priori information about class interdependence, assuming that this interdependence (or the lack thereof) will be discovered in the course of learning.

When employed dogmatically, however, the principle can do more harm than good. The fact that **x** belongs to C_i often has nothing to do with **x** belonging to C_j. If this is the case, insisting on the dependence link between the two (as in Fig. 14.2) can be counterproductive. If most classes are mutually independent, the upper layer may actually exhibit better classification performance than the lower because the newly added attributes (the classes obtained from the upper layer) are *irrelevant*—and as such will hurt induction.

Proper understanding of this issue will instruct our choice of the baseline learner. Some approaches, such as decision trees or WINNOW, are capable of eliminating irrelevant attributes, thus mitigating the problem.

14.4.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- How are interdependent classes addressed by *stacking*? Discuss the induction phase as well as the classification phase.
- When will *stacking* outperform *binary relevance* and/or a *classifier chain*?
- Under what circumstances will you prefer *binary relevance* to *stacking*?

14.5 Note on Hierarchically Ordered Classes

In some domains, the class interdependence is more complicated than in the cases considered so far. Our induction techniques then have to be modified accordingly.

Example Figure 14.3 shows a small part of a class hierarchy that could have been suggested by a specialist preparing the data for machine learning. Each node in the graph represents one class.

The topic is classification of text documents. The hierarchy is interpreted in a way reminiscent of decision trees. To begin, some documents may belong to the class `machine learning`. A solid line emanating from the corresponding node represents "yes," and the dashed line represents "no." Among those documents that do belong to `machine learning`, some deal with `decision trees`, others with k`-NN classifiers`, and so on (for simplicity, most sub-classes are omitted here).

In the picture, the relations are represented by arrows leading from parent nodes to child nodes. A node can have more than one parent, but a well-defined hierarchy should avoid loops. The data structure of this kind is called a *directed acyclic graph*. In some applications, each node (except the root node) has one and only one parent. This more constrained structure is known as a *generalization tree*.

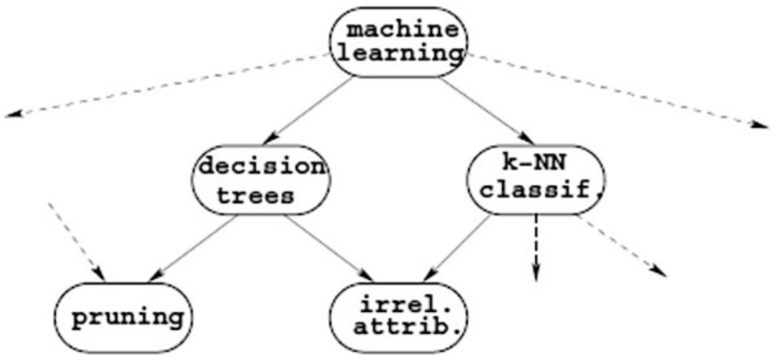


Fig. 14.3 The classes are sometimes hierarchically organized in a way known to the engineer in advance

Induction in "Hierarchical" Domains Induction of hierarchically ordered classes is organized in a way reminiscent of the strategy used in the context of *binary relevance*. For each node, the corresponding training set is constructed, and from this training set, the baseline learner induces a classifier. The most common approach proceeds in a top-down manner; the output of the parent class instructs the choice of the examples for the induction of a child class.

Here is how to do this in a domain where the classes are organized by a generalization tree. First, the entire original training set is used for the induction of the class located at the root of the tree. Next, the training set is divided into two parts, one containing training examples that belong to the root class and the other containing training examples that do *not* belong to this class. The lower-level classes are then induced only from the relevant training sets.

Example In the problem from Fig. 14.3, the first step is to induce a classifier for the class `machine learning`. Suppose that the original training set consists of the seven examples shown in Table 14.2. The labels of those examples are then used to decide which examples to include in the training sets for the induction of the child classes. For instance, note that only positive examples of `machine learning` are included in the training sets for `decision trees` and `k-NN classifiers`. Conversely, only negative examples of `machine learning` are included in the training set for the induction of `programming`.

Two Difficulties The induction process is not as simple as it appears. The first problem is, again, *error propagation*. Suppose an example represents a text document dealing with `circuit analysis`. If this example is mistakenly classified as `machine learning`, the classifier, hereby misled, will pass this incorrect information on to the next classifiers, thus propagating the error down to lower levels.[2]

[2]The reader has noticed that the issue is similar to the one we encountered in the section dealing with classifier chains.

Table 14.2 Hierarchically ordered classes. Some of the lower-level training sets contain only examples for which the parent classifier returned the positive class; in others, only those for which the parent classifier returned the negative class

	machine learning
ex_1	**pos**
ex_2	**pos**
ex_3	**pos**
ex_4	**pos**
ex_5	**neg**
ex_6	**neg**
ex_7	**neg**

decision trees	
ex_1	1
ex_2	1
ex_3	0
ex_4	0

k-NN	
ex_1	0
ex_2	0
ex_3	1
ex_4	1

programming	
ex_5	1
ex_6	0
ex_7	0

Another complication is that the training sets associated with the individual nodes in the hierarchy are almost always heavily *imbalanced*; appropriate measures have to be taken such as undersampling or oversampling.

Where Does the Class Hierarchy Come From? In some rare applications, the complete class hierarchy is available right from the start, having been created manually by the customer who has the requisite background knowledge about the concrete domain. This is sometimes the case applications related to text categorization.

Caution is called for, though. Customers are not infallible, and the hierarchies they develop often miss important details. They may too subjective—with consequences similar to those explained in the context of classifier chains. In some domains, only parts of the hierarchy are known. The engineer then has to find a way of incorporating this partial knowledge in the *binary relevance* framework discussed earlier.

14.5.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Give an example of a domain with hierarchically ordered classes.

- Explain the training-set dividing principle used for induction of hierarchically ordered classifiers.
- What are the most commonly encountered difficulties in the induction of hierarchically ordered classes?

14.6 Aggregating the Classes

This approach makes sense only if there are only a few classes. For instance, the number of all class-label combinations in a domain with three classes cannot exceed seven, assuming that each example is labeled with at least one class.

Creating New Training Sets In a domain with only a few classes, a sufficiently large training set is likely to contain a sufficient number of representatives for each class combination, which makes it possible to treat each such combination as a separate class. The approach is similar to those we have already seen: from the original training set, T, new training sets, T_i, are created, and from each, a classifier is induced by the baseline learner. In *class aggregation*, each T_i represents one combination of class labels, say, C_2 AND C_4.

When, in the future, some example **x** is to be classified, it is presented to all of these classifiers in parallel.

Illustration The principle is shown in Table 14.3. Here, the total number of classes in the original training set, T, is three. Theoretically, the total number of class

Table 14.3 In a domain with a manageable number of class-label combinations, it is often possible to treat each combination as a separate class

	classes
ex$_1$	C_1, C_2
ex$_2$	C_2
ex$_3$	C_1, C_3
ex$_4$	C_2, C_3
ex$_5$	C_1

C_1		C_2		C_1 AND C_2		C_1 AND C_3		C_2 AND C_3	
ex$_1$	0	ex$_1$	0	ex$_1$	1	ex$_1$	0	ex$_1$	0
ex$_2$	0	ex$_2$	1	ex$_2$	0	ex$_2$	0	ex$_2$	0
ex$_3$	0	ex$_3$	0	ex$_3$	0	ex$_3$	1	ex$_3$	0
ex$_4$	0	ex$_4$	0	ex$_4$	0	ex$_4$	0	ex$_4$	1
ex$_5$	1	ex$_5$	0	ex$_5$	0	ex$_5$	0	ex$_5$	0

combinations should be seven. In reality, only five of these are encountered in T because no example is labeled with C_3 alone, and no example is labeled with all three classes simultaneously. We therefore create five tables, each defining the training set for one class combination. Note that this approach deals only with those combinations that have occurred in the original training set. For instance, no future example will be labeled with C_3 alone. This surely *is* a limitation, and the engineer has to consider how to address this issue in a concrete application.

Classification The programmer must also specify what to do in a situation where two or more of these "aggregated" classifiers return 1. In some paradigms, say, in a Bayesian classifier, this is easy because the classifiers quantify their confidence in the outcome they recommend. If two or more classifiers return 1, a master classifier chooses the one with the highest confidence.

The choice is more complicated in the case of classifiers that only return 1 or 0, without offering any information about their confidence in the decision. In principle, one may consider merging the sets of classes. For example, suppose that, for some **x**, two classifiers return 1 and that one of the classifiers is associated with classes C_1, C_3, and C_4 and the other with classes C_3 and C_5. In this event, **x** will be labeled with C_1, C_3, C_4, and C_5.

Note, however, that this may result in **x** being labeled with "too many" classes. The reader already knows that this is likely to give rise to many false positives, the consequence being, for instance, low *precision*.

Alternative Ways of Aggregation In the approach illustrated in Table 14.3, the leftmost table (headed by C_1) contains only one positive label ("1") because only one training example in T is labeled solely with this class. If we want to avoid training sets that are as extremely imbalanced as this, we need a "trick" to improve the class representations in T_i's.

Here is one possibility. In T_i, we will label with 1 each example whose set of class labels in the original T contains C_1. When doing so, we must not forget that $\mathtt{ex}_1$ will thus be labeled as positive also in the table headed with (C_1 AND C_2).

Similarly, we may label with 1 all subsets of the classes found in a given training set. For instance, if **x** is labeled with C_1, C_3, and C_4, we will label this example with 1 in all training sets that represent nonempty subsets of $\{C_1, C_3, C_4\}$. This, of course, improves training only for those combinations that involve one or two classes (rarely more). For larger combinations, the problem persists.

A solution of the last resort will aggregate the classes only if the given combination is found in a sufficient percentage of the training examples. If the combination is rare, the corresponding T_i is not created. True, this means that the induced classifiers will not recognize a certain combination. However, this loss can be tolerated if the affected combination is rarely seen.

Criticism Class aggregation is likely to disappoint in domains where the number of class combinations is high, and the training set is limited. When this happens, some (or many) of the newly created sets, T_i, will contain no more than just a few positive examples and as such will be ill-suited for machine learning: the training sets will be so *imbalanced* that all attempts to improve the situation by minority-class oversampling or majority-class undersampling will fail. For instance, this will be the case when a class combination is represented by a single example.

As a rule of thumb, in domains with a great number of different class labels, where many combinations occur only rarely and some do not occur at all, the engineer will prefer plain *binary relevance* or some of its variations (chaining or stacking). Class aggregation is then to be avoided.

14.6.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Describe the principle of *class aggregation*. Explain separately the induction process and the way the induced classifiers are used to classify future examples.
- What possible variations on the class-aggregation theme do you know?
- What main shortcoming can render this approach impractical in realistic applications?

14.7 Criteria for Performance Evaluation

Performance evaluation in multi-label domains needs to average the results across the individual classes. Let us briefly discuss typical ways of doing so. For simplicity, we will constrain ourselves to *precision, recall,* and F_1. Averaging the other criteria such as *sensitivity, specificity,* and *gmeans* is analogous.

Macro-Averaging The simplest approach, *macro*-averaging, finds the values of the given criterion for each class separately and then calculates their arithmetic average. Let L be the total number of classes. Here are the formulas that calculate macro-*precision*, macro-*recall*, and macro-F_1 from the values of these quantities for the individual classes:

$$Pr^M = \frac{1}{L}\sum_{i=1}^{L} pr_i$$

$$Re^M = \frac{1}{L}\sum_{i=1}^{L} re_i \tag{14.1}$$

$$F_1^M = \frac{1}{L}\sum_{i=1}^{L} F_{1i}$$

Macro-averaging is suitable in domains where each class has approximately the same number of representatives. In some applications, this requirement is not satisfied, but the engineer may still prefer macro-averaging if he or she considers each class to be equally important, regardless of its representation in the training set.

Micro-Averaging In the other approach, *micro*-averaging, each class is weighted by its frequency in the given set of examples. In other words, the performance is averaged over all examples. Let L be the total number of classes. Here are the formulas for micro-*precision*, micro-*recall*, and micro-F_1:

$$Pr^{\mu} = \frac{\sum_{i=1}^{L} N_{TP_i}}{\sum_{i=1}^{L}(N_{TP_i} + N_{FP_i})}$$

$$Re^{\mu} = \frac{\sum_{i=1}^{L} N_{TP_i}}{\sum_{i=1}^{L}(N_{TP_i} + N_{FN_i})} \tag{14.2}$$

$$F_1^{\mu} = \frac{2 \times Pr^{\mu} \times Re^{\mu}}{Pr^{\mu} + Re^{\mu}}$$

Note that F_1^{μ} is calculated from micro-*precision* and micro-*recall* and not from the observed classifications of the individual examples.

Micro-averaging is preferred in applications where the individual classes cannot be treated equally. For instance, the engineer may decide that high performance on dominant classes is more important than failure on classes that are too rare to be of major consequence.

Numeric Example Table 14.4 illustrates these formulas. There are five examples. For each, the middle column lists the correct class labels, and the rightmost column gives the labels returned by the classifier. The reader can see minor discrepancies. For one thing, the classifier has missed some classes, as in the case of class C_3 being missed in example $\mathtt{ex}_3$. On the other hand, the classifier labels some examples with incorrect class labels, such as when example $\mathtt{ex}_1$ is labeled with C_3.

These discrepancies are reflected in the numbers of true positives, false positives, and false negatives and in the calculations of *precisions* and *recalls*. After this, the table shows the calculations of the macro- and micro-averages of these two criteria.

Averaging the Performance Over Examples So far, the true and false positive and negative examples were counted across the classes. However, in domains where an average example belongs to a great many classes, it makes sense to average over the examples.

Table 14.4 Illustration of performance evaluation in multi-label domains

The following table gives, for five testing examples, the known class labels versus the class labels returned by the classifier.

	true	classifier's
	classes	classes
$\texttt{ex}_1$	C_1, C_2	$C_1, C_2, C_3,$
$\texttt{ex}_2$	C_2	$C_2, C_4,$
$\texttt{ex}_3$	C_1, C_3, C_5	$C_1, C_5,$
$\texttt{ex}_4$	C_2, C_3	$C_2, C_3,$
$\texttt{ex}_5$	C_2, C_4	$C_2, C_5,$

Separately for each class, here are the values of true positives, false positives, and false negatives. Next to them are the corresponding values for precision and recall, again separately for each class.

$$\begin{array}{lllll}
N_{TP_1}=2 & N_{FP_1}=0 & N_{FN_1}=0 & Pr_1=\frac{2}{2+0}=1 & Re_1=\frac{2}{2+0}=1\\
N_{TP_2}=4 & N_{FP_2}=0 & N_{FN_2}=0 & Pr_2=\frac{4}{4+0}=1 & Re_2=\frac{4}{4+0}=1\\
N_{TP_3}=1 & N_{FP_3}=1 & N_{FN_3}=1 & Pr_3=\frac{1}{1+1}=0.5 & Re_3=\frac{1}{1+1}=0.5\\
N_{TP_4}=0 & N_{FP_4}=1 & N_{FN_4}=1 & Pr_4=\frac{0}{0+1}=0 & Re_4=\frac{0}{0+1}=0\\
N_{TP_5}=1 & N_{FP_5}=1 & N_{FN_5}=0 & Pr_5=\frac{1}{1+1}=0.5 & Re_5=\frac{1}{1+0}=1
\end{array}$$

This is how the macro-averages are calculated:

$$Pr^M = \frac{1+1+0.5+0+0.5}{5} = 0.6$$

$$Re^M = \frac{1+1+0.5+0+1}{5} = 0.7$$

Here is how the micro-averages are calculated:

$$Pr^\mu = \frac{2+4+1+0+1}{(2+0)+(4+0)+(1+1)+(0+1)+(1+1)} = 0.73$$

$$Re^\mu = \frac{2+4+1+0+1}{(2+0)+(4+0)+(1+1)+(0+1)+(1+0)} = 0.8$$

The principle of the procedure is the same as before. When comparing the true class labels with those returned for each example by the classifier, we obtain the numbers of true positives, false positives, and false negatives. From these, we easily obtain the macro-averages and micro-averages. We only have to keep in mind that the average is not taken over classes, but over examples—thus in macro-averages, we divide the sum by the number of examples, not by the number of classes.

14.7.1 *What Have You Learned?*

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Write down the formulas for *macro*-averaging of *precision*, *recall*, and F_1.
- Write down the formulas for *micro*-averaging of *precision*, *recall*, and F_1. Discuss the difference between *macro*-averaging and *micro*-averaging.
- What is meant by "averaging the performance over examples"?

14.8 Summary and Historical Remarks

- In such domains as text categorization or computer vision, each example can be labeled with two or more classes at the same time. These are so-called *multi-label* domains.
- In *multi-label* domains, traditional paradigms can sometimes be used, but the results may be discouraging unless special precautions have been made. For some approaches, multi-label versions exist, but these are too advanced for an introductory text, especially when good results can be achieved by simpler means.
- The most common approach to *multi-label domains* induces a binary classifier for each class separately and then submits examples to all these classifiers in parallel. This is called *binary relevance*.
- The basic version of *binary relevance* seems to neglect the fact that the classes may not be independent of each other. The fact that an example has been identified as a representative of class C_A may strengthen or weaken its chances of belonging also to class C_B.
- The simplest mechanism for dealing with class interdependence in multi-label domains is the *classifier chain*. Here, the output of one binary classifier is used as an additional attribute describing the example for the next classifier in line.
- One weakness of *classifier chains* is that the user is expected to specify the sequence of classes (perhaps following class subsumption). If the sequence is poorly designed, results disappoint.
- Another shortcoming is known as *error propagation*: an incorrect label given to an example by one classifier is passed on to the next classifier in the chain, potentially misleading the rest of the entire sequence.
- Much safer is two-layered *stacking*. Upper-layer classifiers are induced from examples described by the original attributes, and lower-layer classifiers are induced from examples described by attribute vectors to which the class labels obtained from the upper layer have been added. When classifying an example, the outcomes of the lower-layer classifiers are used.
- Sometimes it is possible to take advantage of known hierarchical order among the classes. Here, too, induction relies on specially designed training sets. Again, the user has to be aware of the dangers of *error propagation*.

- Yet another possibility is *class aggregation* where each combination of classes is treated as a separate higher-level class. A special auxiliary training set is created for each of these higher-level classes.
- The engineer has to pay attention to ways of measuring the quality of the induced classifiers. Observing that each class may experience different classification performances, we need mechanisms for averaging over the classes (or examples). Two of them are popular: micro-averaging and macro-averaging.

Historical Remarks The problem of multi-label classification is relatively new. The first time it was encountered was in the field of text categorization—see McCallum (1999). The simplest approach, the *binary relevance* principle, was employed by Boutell et al. (2004). A successful application of classifier chains was reported by Read et al. (2011), whereas Godbole and Sarawagi (2004) are credited with having developed the *stacking* approach. Apart from the approaches related to *binary relevance*, some authors have studied ways of modifying classical single-label paradigms. The ideas on nearest-neighbor classifiers in multi-label domains are borrowed from Zhang and Zhou (2007) (their technique, however, is much more sophisticated than the one described in this chapter). Induction of hierarchically ordered classes was first addressed by Koller and Sahami (1997). Multi-label decision trees were developed by Clare and King (2001).

14.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

Table 14.5 An example of a multi-label domain

	True classes	Classifier's classes
$\texttt{ex}_1$	C_1	C_1, C_2
$\texttt{ex}_2$	C_1, C_2	C_1, C_2
$\texttt{ex}_3$	C_1, C_3	C_1
$\texttt{ex}_4$	C_2, C_3	C_2, C_3
$\texttt{ex}_5$	C_2	C_2
$\texttt{ex}_6$	C_1	C_1, C_2

14.9.1 Exercises

1. Consider the multi-label training set in the left part of Table 14.5. Show how the auxiliary training sets will be created when the principle of *binary relevance* is to be used.
2. For the same training set, create the auxiliary training sets for *class aggregation*. How many such sets will we need?
3. Draw a schema showing how the problem from Table 14.5 would be addressed by *stacking*. Suppose the examples in the original training set are described by ten attributes. How many attributes will the lower-level classifiers use?
4. Suggest the classifier-chain schema for a domain with the following four classes: `decision trees`, `machine learning`, `classification`, and `pruning`.
5. Returning to the set of examples from Table 14.5, suppose that a classifier has labeled them as indicated in the rightmost column. Calculate the macro- and micro-averages of *precision* and *recall*.

14.9.2 Give it Some Thought

1. Suggest a multi-label application where the *classifier chain* is a reasonable strategy to follow. What would be the main aspect for such a domain?
2. Consider a domain where most training examples are labeled each with only a single class, while a small subset of the examples (say, 5%) is labeled with more than one class. Suggest a machine-learning approach to induce reliable classifiers from such data.
3. Suppose you have a reason to assume that a few classes are marked by strong interdependence, while most of the remaining classes are mutually independent. You are considering *stacking*. What problem may compromise the performance of the induced classifiers? Can you suggest a way to overcome this pitfall?
4. Suppose you are asked to develop machine-learning software for induction from multi-label examples. This chapter has described at least four approaches to choose from. Write down the main thoughts that would guide your choice.
5. Suggest a mechanism that would mitigate the problem of *error propagation* during multi-label induction in a domain with hierarchically ordered classes. Hint: after a testing run, consider "enriching" the training sets by "problematic" examples.

14.9.3 Computer Assignments

1. Write a program that accepts as input a training set of multi-label examples and returns as output the set of auxiliary training sets needed for the *binary relevance* approach.
2. Write a program that converts the training set from the previous question into auxiliary training sets, following the principle of *class aggregation*.
3. Search the web for benchmark domains with multi-label examples. Convert them using the data-processing program from the previous question, and then induce the classifiers using the *binary relevance* approach.
4. Write a program that induces the classifiers using *binary relevance* as in the previous question. In the next step, the program redescribes the training examples by adding to their attribute vectors the class labels required by the lower layer in *classifier stacking*.
5. What data structures would you consider for the input and output data when implementing the *classifier stacking* technique?
6. Write a program that takes as input the values of N_{TP}, N_{TN}, N_{FP}, and N_{FN} for each class and returns micro- and macro-averaged *precision*, *recall*, and F_1.

Chapter 15
Unsupervised Learning

Unsupervised learning seeks to obtain information from training sets in which the examples are not labeled with classes. This contrasts with the more traditional *supervised learning* that induces classifiers from pre-classified data.

Perhaps the most popular unsupervised-learning paradigm looks for groups (*clusters*) of similar examples. The centroids of the clusters can then serve as Gaussian centers for Bayesian classifiers or to define the neurons in RBF networks. There is a lot to learn, from the clusters. For instance, each can be marked by a different set of relevant attributes, and the knowledge of these attributes can help predict unknown attribute values. Clusters can even be used for classification purposes.

Other unsupervised-learning techniques, Kohonen networks and *auto-encoding*, can visualize the data, compress the data, create higher-level attributes from existing ones, fill-in missing information, and even generate artificially created training examples that help increase the chances of supervised learning.

The chapter describes important unsupervised-learning techniques, explains the relevant algorithms, discusses their behaviors, and shows how to benefit from them in practical circumstances.

15.1 Cluster Analysis

The first task in unsupervised learning is *cluster analysis*. The input is a set of examples, each described by a vector of attribute values—but no class labels. The output is a set of two or more clusters formed by the examples.

Groups of Similar Examples Figure 15.1 shows a few examples described by two attributes, x_1 and x_2. An observer will notice that the examples form three or four groups, depending on the subjective "level of resolution."

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_15

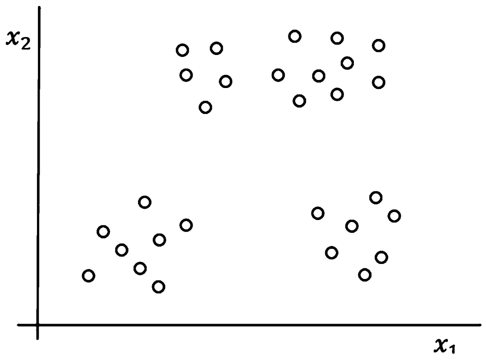


Fig. 15.1 A two-dimensional domain with clusters of examples

To discover such groups in a two-dimensional space by just looking at the examples' graphical representation is easy. The difficulty begins with four or more dimensions where humans can neither visualize the data nor detect the clusters. This is where an algorithm-based analysis can help.

Representing Clusters by Centroids To begin, we need to know how to describe the clusters. A few alternatives exist: it is possible to specify the clusters' locations, their sizes, shapes, and some other aspects. The simplest approach, however, relies on *centroids*.[1] If all attributes are numeric, the centroid is often identified with the averages of the attributes. For instance, suppose that a two-dimensional cluster consists of the following examples: $(2, 5)$, $(1, 4)$, $(3, 6)$. In this case, the centroid is defined by vector $(2, 5)$ because the first attribute's average is $\frac{2+1+3}{3} = 2$ and the second attribute's average is $\frac{5+4+6}{3} = 5$.

The averages can be calculated even when the attributes are discrete. provided that we know how to turn them into numeric ones. Here is a simple possibility. If the attribute can acquire three or more different values, we can replace each attribute-value pair with one Boolean attribute (say, `season=fall, season=winter`, etc.). The values of these individual Boolean attributes are then represented by 0 or 1 instead of *false* and *true*.

For instance, `fall` can then be described as $(0, 0, 1, 0)$. The attentive reader will recall, in that Chap. 6, this was called *one-hot* representation.

What Should the Clusters be Like? Clusters should not overlap each other: each example must belong to one and only one cluster. Within the same cluster, the examples should be relatively close to each other, and they should be distant from the examples belonging to the other clusters.

An important question is how many clusters the available data form. In Fig. 15.1, we noticed that the human observer will discern either three or four clusters.

[1]Machine learning professionals avoid the term "center" which might imply mathematical properties that are irrelevant for the specific needs of cluster analysis.

However, we are not limited to just these two possibilities. At one extreme, the entire training set can be thought of as forming one big cluster; at the other extreme, each example can be seen as representing its own single-example cluster.

Practical implementations often side-step the problem by asking the user to supply the number of clusters by way of an input parameter. Sometimes, however, machine learning is expected to determine the number automatically.

Measuring Distances Algorithms for cluster analysis need a mechanism to evaluate the distance between an example and a cluster. If the cluster is described by its centroid, the Euclidean distance between the two vectors (the example and the centroid) offers a good way of doing so. The reader will recall, however, what Chap. 13 said about situation where Euclidean distance can be misleading.

There is no harm in a little revision. Euclidean distance may be inconvenient in the case of discrete attributes, but we know how to deal with the problem. Also concrete scaling plays a role: distances will change considerably if we switch from feet to miles. For this, some attribute normalization is needed. Much more serious, however, is a situation where each attribute represents a different quantity, which renders geometric somewhat meaningless: a four-year difference in `age` is hard to compare with a 4-foot difference in `height`.

In the context of cluster analysis, distance-related issues tend to be less serious than they were in k-NN classifiers. Most of the time, we can get around the difficulties by normalizing all attribute values to the unit interval, $x_i \in [0, 1]$. We will return to normalization in Sect. 15.2.

General Formula for Distances When the examples are described by a mixture of numeric and discrete attributes, we often rely on the sum of the squared distances along corresponding attributes. More specifically, the following expression is recommended (here, n is the number of attributes):

$$d_M(\mathbf{x}, \mathbf{y}) = \sqrt{\Sigma_{i=1}^{n} d(x_i, y_i)} \tag{15.1}$$

In this formula, we use $d(x_i, y_i) = (x_i - y_i)^2$ for continuous attributes. For discrete attributes (including Boolean attributes), we put $d(x_i, y_i) = 0$ if $x_i = y_i$ and $d(x_i, y_i) = 1$ if $x_i \neq y_i$.

Which Cluster Should an Example Belong to? Suppose that each example is described by vector $\mathbf{x}$, and that each cluster is defined by its centroid—which, too, is an attribute vector.

Suppose there are N clusters with centroids denoted by $\mathbf{c}_i$, where $i \in (1, N)$. The example $\mathbf{x}$ has a certain distance $d(\mathbf{x}, \mathbf{c}_i)$, from each centroid. If $d(\mathbf{x}, \mathbf{c}_P)$ is the smallest of these distances, we expect that $\mathbf{x}$ will find itself in cluster $\mathbf{c}_P$.

For instance, suppose that we use the Euclidean distance, and that there are three clusters. If the centroids are $\mathbf{c}_1 = (3, 4)$, $\mathbf{c}_2 = (4, 6)$ and $\mathbf{c}_3 = (5, 7)$, and if the example is $\mathbf{x} = (4, 4)$, then the Euclidean distances are $d(\mathbf{x}, \mathbf{c}_1) = 1$, $d(\mathbf{x}, \mathbf{c}_2) = 2$, and $d(\mathbf{x}, \mathbf{c}_3) = \sqrt{10}$. Since $d(\mathbf{x}, \mathbf{c}_1)$ is the smallest of the three values, we conclude that $\mathbf{x}$ should belong to $\mathbf{c}_1$.

Benefit 1: Estimating Missing Values The knowledge of the clusters can help us estimate missing attribute values. Returning to Fig. 15.1, we notice that if the value of x_1 is very low, the example probably belongs to the bottom-left cluster. In this case, also x_2 is likely to be low because such is the case in all examples in this cluster. This aspect tends to be even more strongly pronounced in realistic data described by multiple attributes,

In example description, some attribute values may not be known. Section 11.5 proposed to replace a missing value by the average or by the value most frequent in the training set. An estimate by the average or by the most frequent value of the *the example's cluster* is more reliable than that because it averages over relevant data, rather than over all data.

Benefit 2: Reducing the Size of RBF Networks and Bayesian Classifiers Cluster analysis can assist Bayesian learners and radial-basis-function networks. The reader will recall that these paradigms operate with centers.[2] In the simplest implementation, the centers are identified with the attribute vectors of the individual examples. In domains with millions of examples, however, this leads to impractically big classifiers. The engineer then prefers to divide the training set into N clusters, and to identify the Gaussian centers with the centroids of these clusters.

Benefit 3: Clusters as Classifiers Clusters may be useful in classification, too. In many domains, all or almost all examples in a cluster belong to the same class. In this event, it is a good idea to identify each cluster with its dominant class. When a future example, $\mathbf{x}$, is to be classified, it is enough to find the nearest cluster, and then to label $\mathbf{x}$ with this cluster's class.

15.1.1 *What Have You Learned?*

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- What is the defining difference between unsupervised learning and supervised learning?
- What is a cluster? Define a way to describe clusters. How can we measure the distance between an example and a cluster? What possible flaws in these distance do have to be aware of?
- Discuss the three main benefits that the knowledge of clusters can bring.

[2]The section on RBF networks denoted these centers by μ_i's.

15.2 Simple Clustering Algorithm: *k*-Means

Perhaps the simplest algorithm to detect clusters of examples in the training set is known as *k-means*. The k in its name refers to the requested number of clusters—a parameter whose value is supplied by the user.

K-Means Algorithm In the pseudo-code in Table 15.1, the first step creates k initial clusters such that each example finds itself in one and only one cluster (we will have more to say about initialization presently). After this, the coordinates of all centroids are calculated as arithmetic averages of the attributes of the examples within the cluster.

In the next step, *k-means* investigates one example at a time, calculating its distances from all centroids. The nearest centroid then defines the cluster to which the example should belong. If the example already *is* that cluster, nothing needs to be done; otherwise, the example is transferred from the current (wrong) cluster to the new (correct) one. After the relocation, the centroids of the two affected clusters (the one that lost the example, and the one that gained it) are recalculated. The procedure is illustrated by the single-attribute domain from Fig. 15.2. Here, two examples find themselves in the wrong clusters and are therefore relocated. After each example transfer, the vertical bars separating the clusters change their locations, and so do the centroids,

Termination The good thing about the algorithm from Table 15.1 is that the process is guaranteed to reach the state where each example finds itself in the nearest cluster. From this moment on, no further transfers are needed. The clusters do not overlap. Since the goal is usually achieved in a manageable number of steps, the process simply stops when no example has been relocated in the whole epoch.

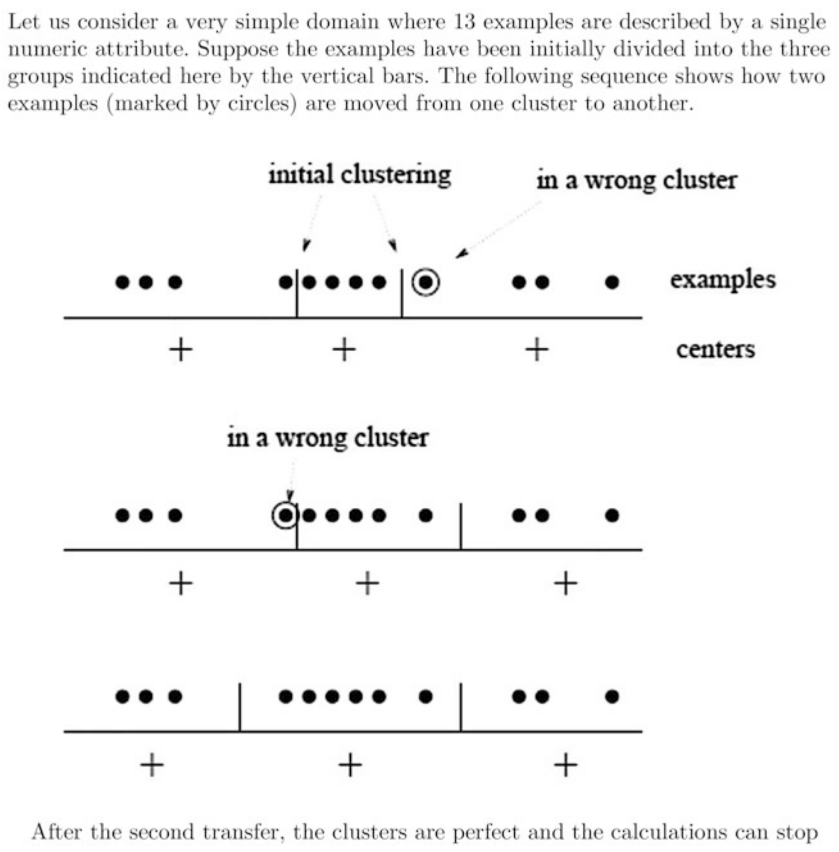
Numeric Example In Table 15.2, a set of nine two-dimensional examples has been randomly divided into three groups (because the user specified $k = 3$), each

Table 15.1 Pseudo-code of *k-means* clustering

```
Input: a set of examples without class labels
       user-set constant $k$

1. Create $k$ initial clusters. For each, calculate the coordinates of its centroid, $C_i$, as the numeric
   averages of the attribute values in the examples in the cluster.
2. Choose an example, $\mathbf{x}$, and find its distances from all centroids. Let $j$ be the index of the
   nearest centroid.
3. If $\mathbf{x}$ already finds itself in the $j$-th cluster, do nothing. Otherwise, move $\mathbf{x}$ from its current
   cluster to the $j$-th cluster and recalculate the affected two centroids.
4. Unless a stopping criterion has been met, repeat the last two steps for another example.

Stopping criterion: each training example already finds itself in the nearest cluster.
```


Let us consider a very simple domain where 13 examples are described by a single numeric attribute. Suppose the examples have been initially divided into the three groups indicated here by the vertical bars. The following sequence shows how two examples (marked by circles) are moved from one cluster to another.

After the second transfer, the clusters are perfect and the calculations can stop

Fig. 15.2 The *k-means* procedure in a domain with a single numeric attribute.

containing the same number of examples. The table also provides the centroids for each group. *K-means* goes through these examples systematically, one by one. Suppose it is now considering group-2. For each example, its distance from each centroid is calculated. It turns out that the first example from group-2 already finds itself in the right cluster. However, the second example is closer to group-1 than to group-2 and therefore has to be transferred from its original cluster to group-1. After this, the affected centroids are recalculated.

Normalizing Attributes to Unit Intervals The reader will recall that, when discussing k-NN classifiers, Chap. 3 argued that inappropriate attribute scaling will distort distances between attribute vectors. The same problem has to be considered in cluster analysis. It is a good idea to normalize the vectors so that all numeric attributes have values from the same unit interval, [0, 1].

Table 15.2 Illustration of the *k-means* procedure in a domain with two attributes

The table below contains three imperfect initial groups of vectors. The task is to find better clusters by the *3-means* procedure.

	Group-1	Group-2	Group-3
	(2, 5)	(4, 3)	(1, 5)
	(1, 4)	(3, 7)	(3, 1)
	(3, 6)	(2, 2)	(2, 3)
Centroids:	(2, 5)	(3, 4)	(2, 3)

Let us pick the first example in group-2. The Euclidean distances between this example, (4, 3), and the centroids of the three groups are $\sqrt{8}$, $\sqrt{2}$, and $\sqrt{4}$, respectively. This means that the centroid of group-2 is the one nearest to the example. Since this is where the example already is, *k-means* does not do anything.

Let us now proceed to the second example in group-2, (3, 7). In this case, the distances are $\sqrt{5}$, $\sqrt{9}$, and $\sqrt{17}$, respectively. Since the centroid of group-1 has the smallest distance from (3, 7), the example is moved from group-2 to group-1. After this, the averages of the two affected groups have to be recalculated.

Here are the new clusters with the new centroids:

	Group-1	Group-2	Group-3
	(2, 5)	(4, 3)	(1, 5)
	(1, 4)	(2, 2)	(3, 1)
	(3, 6)		(2, 3)
	(3, 7)		
Centroids:	(2.25, 5.25)	(3, 2.5)	(2, 3)

The process continues as long as any example transfers are needed.

Note: for simplicity, normalization was ignored here.

The simplest way of doing so is to determine for the each attribute its maximum (MAX) and minimum (MIN) value in the training set. Then, each value of this attribute is recalculated by the following formula:

$$x = \frac{x - MIN}{MAX - MIN} \tag{15.2}$$

As for Boolean attributes, their values can simply be replaced with 1 and 0 for *true* and *false*, respectively. Finally, an attribute that acquires n discrete values (such as `season`, which has four different values) can be replaced with n Boolean attributes, one for each value—and, again, for the values of these Boolean attributes, 1 or 0 are used as in the case of the *one-hot* representation that we already know from Sect. 6.4,

Computational Consequences of Initialization To reach its goal, *k-means* has to make a certain number of transfers of examples from wrong clusters to the right ones. How many such transfers are needed depends on the contents of the initial clusters. Theoretically speaking, we can imagine a situation where the randomly generated initial clusters are already perfect, and not a single example needs to be moved. Of course, this is an extreme, but the message is clear: if the initialization is "lucky," fewer transfers are needed than in the case of an "unlucky" initialization. A better starting point means that the solution is found sooner.

How to Initialize In some domains, we can take advantage of some background knowledge about the problem at hand. For instance, seeking to create initial clusters in a database of a company's employees, the data analyst may speculate that it makes sense to group them by their age, salary, or some other intuitive criterion, and that the groups thus obtained will make good initial clusters.

In other applications, however, no such guidelines exist. The simplest procedure then picks k random training examples and regards them as *code vectors* to define initial centroids. The initial clusters are then created by associating each of the examples with its nearest code vector.

Another Problem with Initialization There is another issue, and quite serious. The thing is, the *composition* of the resulting clusters (once *k-means* has completed its work) may depend on initialization. Choose a different set of initial code vectors, and the technique may generate a different set of clusters.

The point is illustrated in Fig. 15.3. Suppose the user wants two clusters ($k = 2$). If he chooses as code vectors the examples denoted by **x** and **y** then the initial clusters created with the help of these two examples are already perfect.

The situation changes when we choose for the code vectors examples **x** and **z**. In this event, the two initial clusters will have a very different composition, and *k-means* is likely to generate a different set of clusters. The phenomenon will be more pronounced if there are "outliers," examples that do not apparently belong to any of the two clusters.

Summary of the Shortcomings The good thing about *k-means* is that it is easy to explain and easy to implement. This simplicity comes at a price, though. The technique is sensitive to initialization; the user is expected to provide the number of clusters (which he may not be able to do without just guessing); and, as we will see,

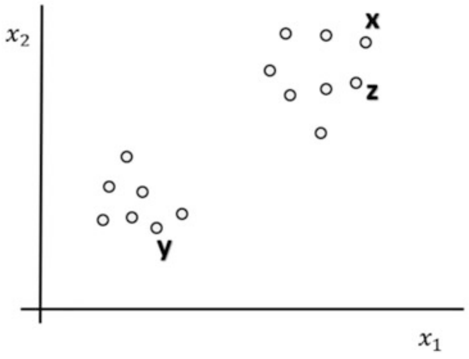


Fig. 15.3 Suppose $k = 2$. If the code vectors are [**x,y**], the initial clusters for k-means will be different then when the code vectors are [**x,z**]

some clusters can never be discovered, in this manner. The next sections will take a look at some techniques to overcome these shortcomings.

15.2.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Describe the principle of the *k-means* algorithm. What can you say about the termination criterion?
- What are the consequence if we do not normalize the training set? Write down the simple normalization formula.
- Describe some methods for the initialization of *k-means*. What are the main consequences of good or bad initialization?

15.3 Advanced Versions of k-Means

The previous section described the baseline version of *k-means* and pointed out its main weaknesses: sensitivity to initialization, and the fact that the user has to submit the requested number of clusters. Let us take a look at some improvements meant to address these shortcomings. First of all, however, we need a performance criterion.

Quality of a Set of Clusters If we want to compare the quality of alternative results of a clustering algorithm, we need objective criteria. One such criterion is built around the original intention to minimize the average distance between examples and the centroids of their clusters.

Let us denote by $d(\mathbf{x}, \mathbf{c})$ the distance between example $\mathbf{x}$ and the centroid, $\mathbf{c}$, of the cluster to which $\mathbf{x}$ belongs. If all attributes are numeric, and if they all have been normalized to unit intervals, then $d(\mathbf{x}, \mathbf{c})$ can be either the Euclidean distance or the more general Eq. 15.1. The following formula (in which *SD* stands for *summed distances*) sums the distances of all examples from their clusters' centroids. Here, $\mathbf{x}_i^{(j)}$ denotes the i-th example in the j-th cluster, K is the number of clusters, n_j is the number of examples in the j-th cluster, and $\mathbf{c}_j$ is the j-th cluster's centroid.

$$SD = \sum_{j=1}^{K} \sum_{i=1}^{n_j} d(\mathbf{x}_i^{(j)}, \mathbf{c}_j) \tag{15.3}$$

A good set of clusters should minimize *SD*. This said, we must not forget that the value obtained by Eq. 15.3 will decrease with the growing number of clusters (in which case, their average size will be smaller), reaching $SD = 0$ in the extreme case when each cluster is identified with one and only one training example. The criterion is thus realistic only if we compare solutions with the same (or at least comparable) numbers of clusters.

Alternative Initializations We already know that the composition of the resulting clusters depends on initialization: different initial code vectors are likely to result in different clusters. It thus makes sense to define two or more sets of initial code vectors, and then apply *k-means* separately to each of them. After this, the quality of all the alternative solutions is evaluated by the criterion specified by Eq. 15.3. The best solution is the one for which the formula results in the lowest value.

Experimenting with Different Values of k One weakness of *k-means* is the requirement that the user should provide the value of k. This is easier said than done because, more often than not, the engineer has no idea how many clusters the data form. Unless more sophisticated techniques are used,[3] the only way out is to try a few alternative values of k, and then pick the one that best satisfies an appropriate criterion (such as the one defined by Eq. 15.3). As already mentioned, the shortcoming of this criterion is that it tends to give preference to small clusters. For this reason, data analysts often normalize the value of *SD* by k, the number of clusters.

Post-processing: Merging and Splitting Clusters The quality of the set of clusters created by *k-means* can often be improved by post-processing techniques that either increase the number of clusters by their splitting or decrease their number by merging some neighbors.

To be more specific, two neighboring clusters can be merged if their mutual distance is small. To decide whether a given distance merits merging, we simply calculate the distance between a pair of centroids, and then compare it with the average cluster-to-cluster distance calculated by the following sum, where $\mathbf{c}_i$ and $\mathbf{c}_j$ are centroids:

$$S = \sum_{i \neq j} d(\mathbf{c}_i, \mathbf{c}_j) \tag{15.4}$$

Conversely, splitting makes sense when the average example-to-example distance within some cluster is high. Concrete solution is not easy to formalize because once we have specified that cluster C is to be split into C_1 and C_2, we need to decide which of C's examples will go to the first cluster and which to the second. Very often, however, it is perfectly acceptable to identify in C two examples with the greatest mutual distance, and then treat them as the code vectors of newly created C_1 and C_2, respectively.

Hierarchical Application of k-Means Another modification relies on recursive calls. The technique begins with running *k-means* for $k = 2$, obtaining two clusters.

[3] One possibility is to visualize the data by the self-organizing feature map. See Sect. 15.5 where the mechanism of visualization is explained and illustrated by an example.

Table 15.3 Pseudo-code of hierarchical *k-means* clustering

```
Input: set $T$ of examples without class labels
       user-specified minimum size of the clusters

Hierarchical-k-means(T)

1. If the size of the cluster is less than the user-specified parameter, stop.
2. Run the $k$-means for $k = 2$, obtaining clusters $C_1$ and $C_2$.
3. Run Hierarchical-k-means($C_1$) and Hierarchical-k-means($C_2$)

Stopping criteria: size of clusters dropped to user-specified minimum.
```

After this, *k-means* is applied to each of these two clusters separately, again with $k = 2$. This continues until an appropriate termination criterion has been satisfied—for instance, the maximum number of clusters specified by the user, or the minimum distance between neighboring clusters or the minimum size of the resulting clusters. Table 15.3.

This hierarchical version side-steps the necessity to provide the required number of clusters because this is here established automatically. The approach is particularly beneficial when the goal is to identify the Gaussian centers in Bayesian classifiers or in RBF networks.

15.3.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Discuss some shortcomings of the *k-means* algorithm, and describe the simple techniques that seek to overcome them.
- Suggest an algorithm to implement the cluster-splitting and cluster-merging techniques described in this section.
- Explain the principle of hierarchically applied *k-means*.

15.4 Hierarchical Aggregation

Just like any other machine-learning technique, *k-means* has its advantages, but also shortcomings. This is why we need an alternative, an approach to turn to in situations where *k-means* fails.

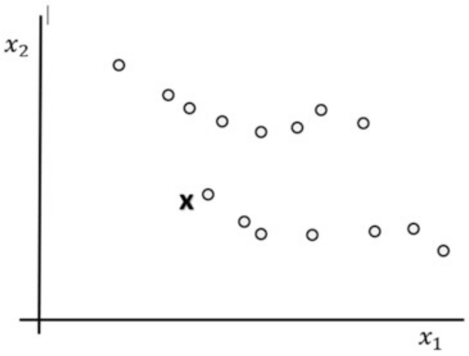


Fig. 15.4 The leftmost example in the bottom cluster is closer to the upper cluster's centroid than to its own. Therefore, *k-means* will not find the best solution

Serious Limitation of k-Means By minimizing the distance between examples and the centroids of the clusters to which these examples belong, *k-means* primarily creates clusters of convex shape. Most of the time, this is what we want—for instance, in the context of Bayesian classifiers and the RBF networks where the clusters help us reduce the number of Gaussian centers.

K-means, however, will do a poor job if the clusters are not convex. Take a look at Fig. 15.4. Here, the leftmost example, **x**, of the bottom cluster is closer to the centroid of the upper cluster, and *k-means* would relocate it accordingly, and yet it is clear that this would not do justice to the nature of the two groups.

To deal with data of this kind, we need another technique.

Alternative Way of Measuring Inter-Cluster Distances Previously, the distance between two clusters was measured by the Euclidean distance between their centroids. In the approach described below, we will do it differently: we will evaluate the distances between all pairs of examples, **[x,y]**, such that **x** comes from the first cluster and **y** from the second. The smallest value among all these example-to-example distances then defines the distance between the two clusters.

A glance at Fig. 15.4 will convince us that, along this new metric, example **x** is closer to the bottom cluster than to the upper cluster, which is what we wanted. What has worsened, though, is computational costs: if N_A is the number of examples in the first cluster, and N_B the number of examples in the second, then $N_A \times N_B$ example-to-example distances have to be evaluated. Most of the time, however, the clusters are not big, and the costs can be tolerated.

Numeric Example For illustration, consider the following two clusters, A and B.

A	B
$\mathbf{x}_1 = (1, 0)$	$\mathbf{y}_1 = (3, 3)$
$\mathbf{x}_2 = (2, 2)$	$\mathbf{y}_2 = (4, 4)$

We calculate the Euclidean example-to-example distances as follows:
$d(\mathbf{x}_1, \mathbf{y}_1) = \sqrt{13}$,
$d(\mathbf{x}_1, \mathbf{y}_2) = \sqrt{25}$,
$d(\mathbf{x}_2, \mathbf{y}_1) = \sqrt{2}$,
$d(\mathbf{x}_2, \mathbf{y}_2) = \sqrt{8}$.
Observing that the smallest among these values is $d(\mathbf{x}_2, \mathbf{y}_1) = \sqrt{2}$, we conclude that the distance between the two clusters is $d(A, B) = \sqrt{2}$.

Hierarchical Aggregation Table 15.4 shows the pseudo-code of the clustering technique known as *hierarchical aggregation*.

In the first step, each example defines its own cluster. This means that in a domain with N examples, we will have N initial clusters. In a series of the subsequent steps, hierarchical aggregation always finds a pair of clusters with the smallest mutual distance along the metric from the previous paragraphs. These clusters are then merged. At an early stage, this typically means to merge pairs of neighboring examples. Later, this results either in adding an example to its nearest cluster or in merging neighboring clusters. Figure 15.5 gives us an idea of how hierarchical aggregation handles the data from Fig. 15.4.

The process continues until an appropriate termination criterion is satisfied. One possibility is to stop when the number of clusters drops below a user-specified

Table 15.4 Pseudo-code of *hierarchical aggregation*

```
Input: a set of examples without class labels

1. Let each example form one cluster. For $N$ examples, this means $N$ clusters, each containing a
   single example.
2. Find a pair of clusters with the smallest cluster-to-cluster distance. Merge the two clusters into
   one, thus reducing the total number of clusters to $N - 1$.
3. Unless a termination criterion is satisfied, repeat the previous step.
```

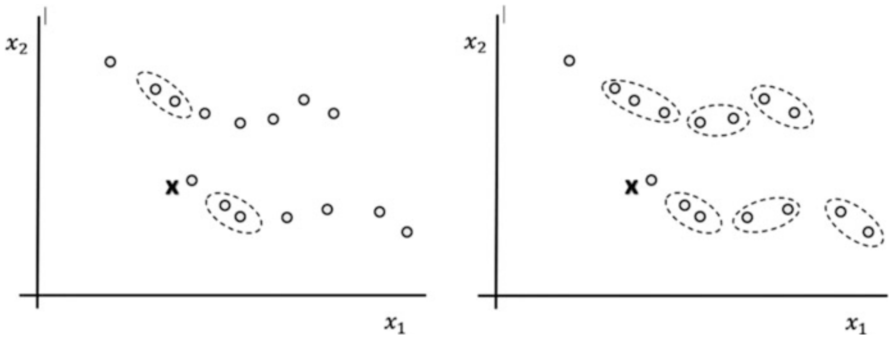


Fig. 15.5 Hierarchical aggregation after the first two steps (left) and after the first nine steps (right). Observe how the clusters are gradually developed

threshold. Alternatively, the program can be asked to finish when the smallest among the cluster-to-cluster exceeds a certain value.

15.4.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- What kind of clusters cannot be detected by the *k-means* algorithm?
- What distance metric is used in *hierarchical aggregation*? What are the advantages and disadvantages of this metric?
- Summarize the algorithm of *hierarchical aggregation*. For what kind of clusters is it particularly suited?

15.5 Self-Organizing Feature Maps: Introduction

Let us now introduce another approach to unsupervised learning, this time borrowing from the field of neural networks. The technique is known as a *self-organizing feature map, SOFM*.[4] Another name commonly used in this context is *Kohonen networks*, to honor its inventor.

Basic Idea Perhaps the best way to explain the nature of SOFM is to use the metaphor of physical attraction. A *code vector*, initially generated by a random-number generator, is subjected to the influences of a series of examples (attribute vectors), each "pulling" the vector in a different direction. In the long run, the code vector settles in a location that is a compromise of all these conflicting forces.

The network consists of a set of *neurons* arranged in a two-dimensional matrix such as the one shown in Fig. 15.6. Each neuron represents a code vector. All code vectors have the same length that is equal to the number of the attributes describing the training examples. At the bottom of the picture is an input attribute vector that is connected to all neurons in parallel. The idea is to achieve by training a situation where neighboring neurons (in the matrix) respond similarly to similar input vectors.

How to Model Attraction Each neuron is represented by a weight vector, $\mathbf{w} = (w_1, \ldots, w_n)$ where n is the number of attributes. If $\mathbf{x} = (x_1 \ldots, x_n)$ is an example, and if $\eta \in (0, 1)$ is a user-specified learning rate, then the individual weights are modified using the following formula:

$$w_i = w_i + \eta(x_i - w_i) \tag{15.5}$$

[4]In statistics, and in neural networks, scientists often use the term *feature* instead of *attribute*.

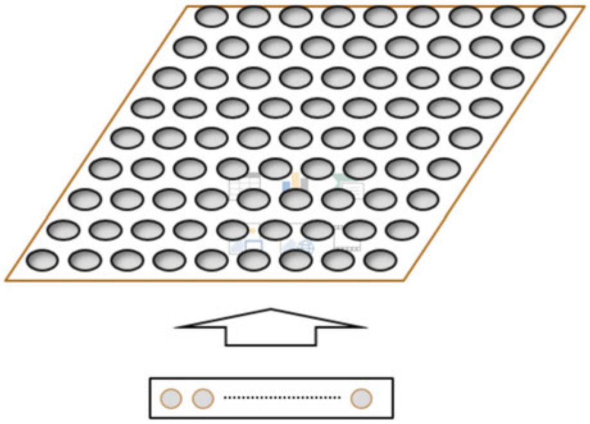


Fig. 15.6 General schema of the Kohonen network. The input vector is presented simultaneously to all neurons in a single the matrix

Note that the i-th weight is increased if $x_i > w_i$ because the term in the parentheses is then positive (and η is always positive). Conversely, the weight is decreased if $x_i < w_i$ because then the term is negative. It is in this sense that we say that $\mathbf{w}$ is *attracted* to $\mathbf{x}$. The strength of this attraction is determined by the learning rate.

Numeric Example Suppose that an example $\mathbf{x} = (0.2, 0.8)$ has been presented, and suppose that the winning neuron has weights $\mathbf{w} = (0.3, 0.7)$. If the learning rate is $\eta = 0.1$, then the weights change in the following way:

$$w_1 = w_1 + \eta(x_1 - w_1) = 0.3 + 0.1(0.2 - 0.3) = 0.3 - 0.01 = 0.29$$
$$w_2 = w_2 + \eta(x_2 - w_2) = 0.7 + 0.1(0.8 - 0.7) = 0.7 + 0.01 = 0.71$$

Note that the first weight originally had a greater value than the first attribute. The force of the attribute's attraction thus reduced this weight. Conversely, the second weight was originally smaller than the corresponding attribute, but the attribute's "pull" has increased it.

Which Weight Vectors Are Modified? The presentation of an example, $\mathbf{x}$, initiates a two-step process. The first step, *competition,* identifies in the matrix the neuron whose weight vector is most similar to $\mathbf{x}$. To this end, the Euclidean distance is used, and smaller distance indicates greater similarity. Once the winner has been established, the second step updates the weights of this winning neuron—and also the weights of all neurons in the winner's physical neighborhood in the matrix.

Note on *Neighborhood* Figure 15.7 illustrates the idea of the neighborhood of the winning code vector $\mathbf{c}_{winner}$. Informally, the neighborhood consists of a set of neurons within a specific physical distance from $\mathbf{c}_{winner}$. The intention is to make sure that the weights of neurons that are physically close to each other are updated in a like manner.

Usually, the size of the neighborhood is not fixed; it is a common practice to reduce it over time as indicated in the right portions of Fig. 15.7. Ultimately, the

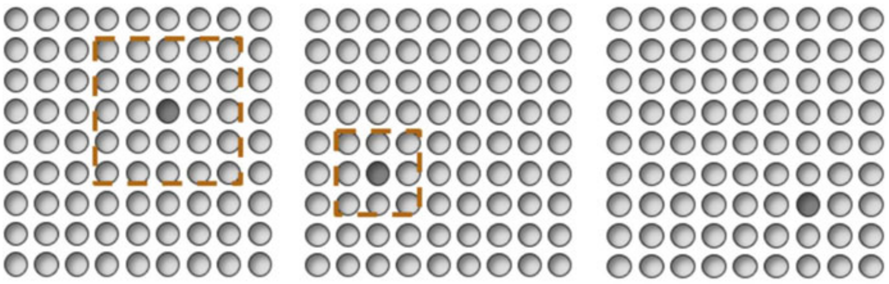


Fig. 15.7 The idea of *neighborhood* in the Kohonen network. After a certain number of epochs, the size of neighborhood is decreased, then again, until it contains only the winning neuron

neighborhood will degenerate to the single neuron that has won the competition. The idea is to start with a coarse approximation that is gradually fine-tuned.

Why Does it Work? The idea is to make sure that neurons physically close to each other in the matrix respond to similar examples. This is why the same weight-updating formula is applied to all neurons in the winner's neighborhood.

15.5.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Describe the general architecture of self-organizing feature maps. Relate the notion of *code vector* to that of a *neuron* in the matrix.
- Explain the two steps of the self-organizing algorithm: competition, and weight adjustment. Comment also on the role of the learning rate, η.
- What is meant by the *neighborhood* of the winning code vector? Is its size always constant?

15.6 Some Details of SOFM

Having outlined the principle, we are ready take a look at some details that are necessary for SOFM's proper functioning.

Normalization For the technique to work, all vectors have to be normalized to unit length (i.e., length equal to 1). This applies both to the vectors describing the examples, and to the weight vectors of the neurons. Fortunately, normalization to unit length is easy. Suppose an example is described by the following attribute vector: $\mathbf{x} = (x_1, \ldots, x_n)$ To obtain unit length, we divide each attribute's value by the length of the original attribute vector, $\mathbf{x}$:

$$x_i := \frac{x_i}{\sqrt{\Sigma_j x_j^2}} \tag{15.6}$$

Numeric Example Suppose we want to normalize the two-dimensional vector $\mathbf{x} = (5, 5)$. The length of this vector is $l(\mathbf{x}) = \sqrt{x_1^2 + x_2^2} = \sqrt{25 + 25} = \sqrt{50}$. Dividing the value of each attribute by this length results in the following normalized version of the vector:

$$\mathbf{x}' = (\frac{5}{\sqrt{50}}, \frac{5}{\sqrt{50}}) = (\sqrt{\frac{25}{50}}, \sqrt{\frac{25}{50}}) = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$$

That the length of $\mathbf{x}'$ equals 1 is easy to verify: using the Pythagorean Theorem, we calculate it as $\sqrt{x_1^2 + x_2^2} = \sqrt{\frac{1}{2} + \frac{1}{2}} = 1$. We can see that the new attribute vector indeed has unit length.

Initialization The first step in SOFM initializes the neurons' weights. Usually, this is done by a random-number generator that chooses the values from an interval that spans equally the positive and negative domains, say, $[-1, 1]$. After this, the weight vector is normalized to unit length as explained above.

For the learning rate, η, small numbers are used, say, $\eta = 0.1$. Sometimes it makes sense to start with a higher value (e.g., $\eta = 0.9$) that then gradually decreases. The point is to modify the weights more strongly at the beginning, to achieve some early approximation. Later, smaller values are used for fine-tuning.

Self-Organizing Algorithm The general principle of self-organizing feature maps is summarized by the pseudo-code in Table 15.5. At the beginning, all examples are normalized to unit lengths. Initial code vectors, too, are created by a random-number generator and then normalized. In the algorithm's main body, training examples are presented one by one. After the presentation of example $\mathbf{x}$, SOFM identifies a neuron whose weight vector, $\mathbf{c}_{winner}$, has the smallest Euclidean distance from $\mathbf{x}$. Then, the weights of $\mathbf{c}_{winner}$ as well as the weights of all neurons in its physical neighborhood are modified by Eq. 15.5, and then re-normalized. The algorithm is run for a predefined number of epochs.[5]

In the course of this procedure, the value of the learning rate is gradually decreased. Occasionally, the size of the neighborhood is reduced: as the training process advances, fewer neurons are affected by the presented examples.

[5]Recall that one epoch means that all training examples have been presented once.

Table 15.5 Pseudo-code of self-organizing feature map (SOFM)

```
Input: set of examples without class labels
       learning rate, $\eta$.
       matrix of neurons, their weights randomly initialized and normalized.

1. Normalize all training examples to unit length.
2. Present example $\mathbf{x}$, and find the neuron, $\mathbf{c}_{winner}$ whose weight vector has the smallest distance
   from $\mathbf{x}$.
3. Modify the weights of $\mathbf{c}_{winner}$ and the weights of the neurons in $\mathbf{c}_{winner}$'s physical neighbor-
   hood. Re-normalize the weight vectors.
4. Unless a stopping criterion is met, present another training example, identify $\mathbf{c}_{winner}$, and
   repeat the previous step.

Comments:

1. $\eta$ usually begins with a relatively high value from (0, 1), then gradually decreases.
2. Every now and then, the size of the neighborhood is reduced.
```

15.6.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- What is the goal of normalization in the context of this section? Write down the normalization formula. Which vectors have to be normalized?
- How is Kohonen's network initialized? What values are used?
- Summarize the algorithm of self-organizing feature maps. Comment on the values for the learning rate and explain the reason for the gradually diminishing *neighborhood*.

15.7 Why Feature Maps?

Let us take a look at the practical benefits of self-organizing feature maps.

Reducing Dimensionality The technique essentially maps the N-dimensional space of the original attribute vectors to the two-dimensional space of the neural matrix. Each example has its winning neuron, and the winner is defined by its two coordinates in the matrix. These coordinates can be regarded as a new pair of attributes that re-describe the original example.

Creating Higher-Level Features Reducing the number of attributes is critical in domains where the number of attributes is prohibitively high, especially if most of these attributes are either irrelevant or redundant. For instance, this is the case in computer vision where each image (i.e., an example) can be described by hundreds of thousands of pixels.

The reader knows that a direct application of, say, perceptron learning to attribute vectors of extreme length is unlikely to succeed: in a domain of this kind, a classifier that does well on a training set tends to fail on testing data. This is because the countless attributes make the VC-dimension so high as to impair learnability—unless the training set is unrealistically large. Advanced machine-learning applications therefore seek to reduce the dimensionality either by attribute selection or by way of mapping the multi-dimensional problem to a smaller space—typically by creating new, higher-level features as functions of the original features.

Algorithm for Feature Reduction We may take this idea one step further. Suppose the training examples are described by 100 attributes, and suppose we have a reason to suspect that some attributes are mutually interdependent, whereas quite a few others are either irrelevant or redundant. The dimensionality thus seems to be unnecessarily high, and any attempt to reduce it is welcome.

One way to do so is by dividing the attribute set into, say, five non-overlapping subsets (20 attributes in each), and then creating five new training sets, each characterizing the examples by a different attribute subset. Each of these training sets is then subjected to SOFM that maps its 20-dimensional space to two dimensions. As a result, we have $5 \times 2 = 10$ new attributes.

Visualization Human brain easily visualizes two- or three-dimensional data, and the mind's eye can "see" the distribution of the examples, the groups they form, and so on. This is impossible when there are more than three attributes.

This is where SOFM can help. By mapping each example onto a two-dimensional matrix, we may visualize at least some of the aspects. For instance, similar attribute vectors are likely to be mapped to the same neuron, or to neurons that are physically close to each other. In this sense, the feature maps serve as visualization tool.

The visualization algorithm is summarized in Table 15.6. Once the SOFM has been trained, each neuron is assigned a counter that is initialized to zero. Then, all training examples are presented to the network, one at a time. After each example presentation, the counter of the winning neuron is incremented.

When this is finished, the network is displayed on the computer screen as in a way indicated by Fig. 15.8: each neuron's shade of grade is determined by the value of the corresponding counter—the more examples for which the neuron was the winner (we say that the winning neuron is *fired* by the example), the darker the shade. In this particular example, the network suggests that the data form two clusters, the black opposing corners of the network.

Initializing the *k*-Means Algorithm Each of the weight vectors in the neural matrix can be regarded as a code vector. The mechanism of SOFM makes it possible

Table 15.6 Data visualization with self-organizing feature map (SOFM)

Input: set of examples without class labels
matrix of neurons

1. Normalize all example vectors to unit lengths.
2. Initialize all neural weights (random values, unit vector lengths).
3. Train the SOFM network.
4. Assign to each vector a counter initialized to 0.
5. Present an example and observe which neuron is the winner. Increment this neuron's counter. Repeat for all training examples.
6. At the end, show the network, with each neuron's shade of gray determined by its counters final value (the higher the value, the darker the neuron).

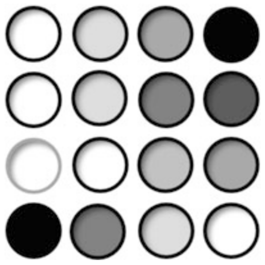


Fig. 15.8 The darker the neuron, the higher the number of examples that have fired it. This picture suggests two clusters and some noise

to find reasonably good values of these vectors—and these can then be used to initialize cluster analysis techniques such as *k-means*. Whether this is practical is another question; SOFM is computationally expensive. Besides, it is not quite obvious whether each neuron should define a cluster, or whether to consider only those clusters that are fired by many training examples (the very dark neurons in Fig. 15.8).

Using SOFM for Classification Purposes Experience with traditional machine-learning paradigms (say, *k*-NN classifiers) has taught us that examples of the same class tend to occupy specific regions in this original feature space. Since SOFT is capable of identifying these regions, it can serve classification purposes.

The classification algorithm is shown in Table 15.7. First, we train the SOFM on examples from which the class labels have been removed. Once this has been completed, we present to the network the training set *with* the class labels. For each neuron, we keep a tally of the classes of the examples for which it was the winner. The neuron is then labeled with the class most common among these examples.

When using the network to classify example **x**, we first decide which of the SOFM neurons is fired by it (which example wins the competition by having the smallest Euclidean distance), and then label **x** by the class associated with this neuron.

Table 15.7 Pseudo-code for using SOFM for classification purposes

```
Learning stage:

1. Train the SOFM using non-classified training examples.
2. Submit all training examples to the induced SOFM. For each neuron, keep a tally of the classes
   of the examples which fired it (for which it was the winner).
3. Label each neuron with the class prevailing among the examples that fired it.

Classification stage:
Submit to SOFM an example, x, and decide which neuron it fires. Label x with the class associated
with this neuron.
```

Follow-up Exercise Once the SOFM has been developed, consider the following experiment. Take the same data that have been used for SOFM training, and label each example with the winning neuron. The resulting training set can then be used by any of the supervised learning from the previous chapters. Induce from the decision tree. Analysis of this decision tree will tell you which attributes are important for different regions of the instance space.

To see the point, recall that each neuron in the Kohonen network represents a cluster, a meaningful region in the instance space. Thanks for the induced decision tree, we now know which attributes matter. For instance, a Kohonen network implemented as a five-by-five matrix consists of 25 neurons. Each of them will appear in at least one leaf node of the tree. The tests along the branches from the root to the leaves labeled with the given neuron inform us about the attributes relevant for the region represented by the neuron. These attributes and the results of their tests may even help us fill-in missing attribute values in future data.

15.7.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- In what way can SOFM help us visualize available data?
- Explain how SOFM can be used to replace the original attributes with new, higher-level features.

15.8 Auto-Encoding

Earlier chapters pointed out that learning from examples described by detailed low-level features is impractical, and they repeatedly called for mechanisms to create meaningful higher-level features. Section 15.7 explained how to use to this end Kohonen networks. Another possibility is *auto-encoding*, AE.

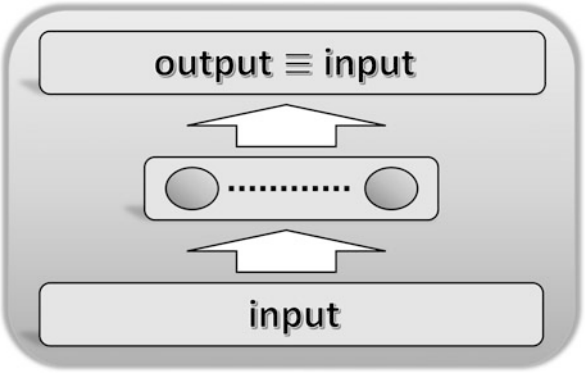


Fig. 15.9 Auto-encoding (AE). Class labels are ignored. At each step, a training example is presented at the input and the output. The goal is to achieve 1-to-1 mapping. The outputs of the hidden layer can serve as higher-level features

Essence of Auto-Encoding Figure 15.9 shows the most common architecture of an auto-encoding network, implemented as a multilayer perceptron. The number of inputs, N, is the same as the number of outputs; importantly, the number of hidden neurons, K, is smaller, $K < N$.

At each step of the training, the same attribute vector is presented at the input and at the output where it serves as the target vector, **t**. Class labels are ignored. The input example is propagated to the output layer, generating an output vector, **y**, that is then compared with **t**. The difference between **y** and **t**, in the form of the mean squared error, MSE, is then backpropagated through the network, and used for weight modification in the same way as in Chap. 6. The goal is to achieve a 1-to-1 mapping from the input to the output.

Perfect mapping with zero MSE is almost impossible to achieve, but this does not matter much. The main idea is to get as close to perfection as permitted by the given domain and by the network's dimensions.

AE Creates Higher-Level Features Once the training by backpropagation has been completed, the outputs of the hidden layer can serve as the higher-level features. Thanks to the requirement that $K < N$, there are fewer of these features than was the number of the original attributes. In this sense, AE causes *data compression*.

Let the k-th original attribute be denoted by x_k, let the weight of the lower-layer link leading from the k-th input to the j-hidden neuron be denoted by $w_{kj}^{(2)}$, and let $f(\sum)$ represent the activation function (e.g., `sigmoid`). The j-th new feature is then defined by the following formula:

$$h_j = f(\sum_j w_{kj}^{(2)} x_k)$$

From the new features, the original ones can be again recovered by multiplying h_j's with the known weights $w_{jk}^{(1)}$.

Benefits of the New Features By re-describing the training examples in terms of the new features, h_j's, we reduce the dimensionality of the instance space (because $K < N$). This means smaller VC-dimension, and therefore improved learnability (recall the conclusions of computational learning theory from Chap. 7).

The circumstance that the danger of overfitting is now lower can be verified by a simple experiment: induce two decision trees, one using the original attributes, and the other using the features obtained from the hidden layer of the AE-network. Chances are high that the second decision tree will give better results on independent testing data.

Size of the Hidden Layer As everywhere, in machine learning, the decision about the number, K, of hidden neurons hinges on a trade-off. Let N be the number of the original attributes. If K is close to N, the reduction of the instance space is insignificant, and the entire AE adventure is hardly worth the trouble. If, conversely, K is too small, then the network will lack flexibility, and its training will fail to achieve even crude approximation of the coveted 1-to-1 mapping.

The ideal size can be established experimentally: we want to find the minimum K that still permits reasonably accurate 1-to-1 mapping. In some domains, the redundancy in the attributes is so high that considerable reduction is possible; in others, the existing attributes are all important, and the higher-level features may destroy some critical information. When preparing the experiments, however, do not forget that they can be computationally expensive: for each value of K, yet another MLP has to be trained.

Frequently Overlooked Detail If the transfer function of the output neurons is `sigmoid`, the output is constrained to the open interval $(0, 1)$. This means that we can never even approximate the 1-to-1 mapping if the original attributes acquire values outside this interval. This is easily avoided if we normalize the original attributes into the unit interval, $(0, 1)$. This is achieved by the mechanism we know from Chap. 3. If MIN and MAX are the minimum and maximum values of a given attribute in the training set, the attribute is normalized by recalculating each of its values with the following formula:

$$x = \frac{x - MIN}{MAX - MIN}$$

Multilayer Version of AE The reader will recall that MLPs sometimes do not allow good mapping unless the hidden layer is large, perhaps much larger than the length of the original attribute vectors (Chap. 6). This, of course, beats the purpose of auto-encoding. A possible solution relies on using more layers, as shown in Fig. 15.10, with the hidden layers becoming progressively shorter. The new features are those observed at the outputs of the smallest layer. This reduces the size of the feature space in a step-wise manner.

The engineer should not become carried away by the seemingly unlimited opportunities offered by multilayer AE. The costs of having to train many alternative architectures may easily outweigh the desired benefits.

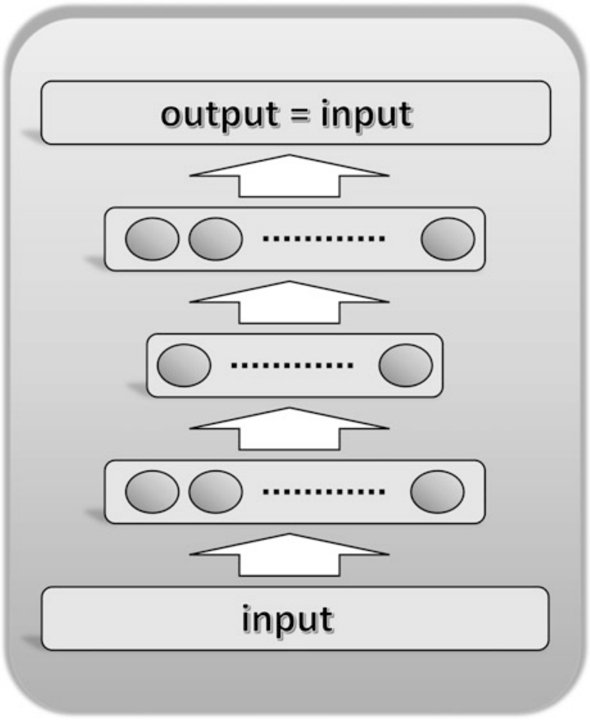


Fig. 15.10 Auto-encoding can involve more than just two layers. The new features are those that are output by the hidden layers. Experiments may reveal which set of new features is best

15.8.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- Explain the principle of *auto-encoding*. What is the main motivation behind it and what is the main benefit? Why should the input data be normalized?
- Discuss the trade-offs involved in the various sizes of the hidden layer.
- Comment on the possibilities offered by multilayer *auto-encoding*.

15.9 Why Auto-Encoding?

Now that we understand the principles of *auto-encoding*, let us take a closer look at how to benefit from it when pursuing diverse machine-learning goals.

Higher-Level Features Some domains are marked by very high numbers of low-level features that inflate the problem's VC-dimensions in exchange for unnecessary detail. These features thus necessitate impractically large training sets. *Auto-*

encoding gives the engineer an opportunity to create a smaller number of more meaningful higher-level features.

Creating Artificial Data In previous chapters, we encountered the need to synthesize artificial examples to be added to the training set needed for supervised learning. This was the case in domains with heavily imbalanced class representation where one of the techniques for dealing with the problem was *minority-class oversampling*. In its simplest version, this approach simply assumed that some examples from the less frequent class should be copied. In a more sophisticated version, the copies were modified by added noise. Auto-encoding offers yet another way of creating these new examples.

The idea is inspired by the observation that the "1-to-1 mapping" can only be imperfect: when we present the trained AE-network with a training example and forward-propagate it, the output is somewhat different from the original example. If we want to create synthetic examples similar to the existing ones, we can train the network with the original training set until the mapping is reasonable (though not perfect); after this, we present each training example at the input, forward-propagate, and thus obtain its imperfect replica.

In this way, we double the number of the training examples. The procedure can then be applied recursively: the new training set (twice the size of the original one) is used to re-train the AE-network, and the result is used to create for each of the new training examples its corrupted copy.

Correcting Imperfect Examples In some domains, the 1-to-1 mapping obtained after the AE-training is almost perfect. In that case, the resulting network can be used to fill-in the likely values for missing attributes.

The principle is similar to the one from the previous paragraph. Once the training has been completed, present at the networks' input an incomplete example where the missing values are replaced either with zeros or with the average values observed in the training set. After forward propagation, the output suggests the likely correct values.

The same approach can perhaps be used to reduce noise.

Combining Supervised and Unsupervised Learning Some domains are marked by a great many examples, only a small proportion of which have been classified. Classical supervised machine learning use for classifier induction only examples that have been labeled, ignoring the others. This, however, may squander potentially useful information. At the very least, the unclassified example can be used for extraction of useful patters—or features.

Consider a domain with 10^6 examples, of which only 10^4 have class labels. If the examples are described by many attributes, we can apply auto-encoding to the many non-classified examples, and obtain in this manner a much smaller feature set.

We then apply some supervised machine-learning technique to the 10^4 classified examples that have been re-described by the new features. Given that the number of these new features is smaller, and that they have been created based on the analysis of a million non-classified examples, we can expect better results than those obtained using the original attributes.

15.9.1 What Have You Learned?

To make sure you understand the topic, try to answer the following questions. If needed, return to the appropriate place in the text.

- How can *auto-encoding* create artificial examples from existing ones?
- How can *auto-encoding* address the problem of missing attribute values?
- How can *auto-encoding* be used in large domains where only a small proportion of the examples have been labeled?

15.10 Summary and Historical Remarks

- *Unsupervised learning* seeks to induce useful knowledge from examples that do not have class labels.
- The simplest unsupervised-learning task is *cluster analysis*. The goal is to find a way to naturally divide the training set into groups of similar examples.
- A popular approach to *cluster analysis* is *k-means*. The technique accepts one example at a time, and evaluates its distances from centroids of all clusters. If the example finds itself in a wrong cluster, it is transferred to a better one. The technique converges in a finite number of steps.
- The quality of the clusters discovered by *k-means* is sensitive to initialization and to the user-specified value of k. Methods to improve this quality by subsequent merging and splitting, and by alternative initializations are sometimes employed. Another possibility is *hierarchical k-means*.
- In some domains, the shapes of the data clusters make it impossible for *k-means* to a good job. In this event, we may prefer some other technique such as *hierarchical aggregation* that creates the clusters in a bottom-up manner, always merging the clusters with the smallest mutual distance.
- In *hierarchical aggregation*, it is impractical to measure the distance between clusters by the distance between their centroids. Instead, we use the minimum distance between **[x,y]** where **x** belongs to one cluster and **y** to the other.
- The technique *k-means* is sensitive to the initial *code vectors*—not only to their locations, but especially to their number, k. One way to address this issue is to use *self-organizing feature maps* that are capable of visualizing the data.
- As an added bonus, *self-organizing feature maps* are capable of converting a high-dimensional feature space onto only two attributes. They can thus be used for feature-reduction purposes. They can even be easily converted to classifiers.

- Another beneficial technique in unsupervised learning is *auto-encoding*. Its primary motivation used to be data compression: reduction of the size of the instance space by creating meaningful higher-level features.
- *Auto-encoding* can be used for the synthesis of artificial training examples, for preparing better features in domains where a great percentage of examples are not labeled with classes, and (sometimes) even for noise suppression.
- Another possible application of *auto-encoding* is in domains with very large source of examples of which only a small proportion have been labeled with classes.

Historical Remarks The problems of cluster analysis have been studied since the 1960s. The *k-means* algorithm was described by McQueen (1967) and hierarchical aggregation by Murty and Krishna (1980). The idea of merging and splitting clusters (not necessarily those obtained by *k-means*) was studied by Ball and Hall (1965). The technique of SOFM, self-organizing feature maps, was developed by Kohonen (1982). In this book, however, only its very simple version was presented. The oldest scientific paper dealing with *auto-encoding* seems to be Hinton and Zemel (1994). Some say that the idea was first proposed by the French PhD dissertation by LeCun (1987).

15.11 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

15.11.1 Exercises

1. Look at the three initial clusters of 2-dimensional vectors in Table 15.8. Calculate the coordinates of their centroids.
2. Using the Euclidean distance, decide whether all examples from group 1 are where they belong. You will realize that one of them is not. Move it to a more appropriate group and recalculate the centroids.
3. Normalize the examples in Table 15.8 using Eq. 15.2.

Table 15.8 Initial set of three clusters

Group 1	Group 2	Group 3
(1, 4)	(4, 3)	(4, 5)
(3, 6)	(6, 7)	(3, 1)
(3, 5)	(2, 2)	(2, 3)

4. Suppose the only information you have is the set of the nine training examples from Table 15.8, and suppose that you want to run the *k-means* algorithm for $k = 3$ clusters. What will be the composition of the three initial clusters if your code vectors are $(1, 4)$, $(3, 6)$, and $(3, 5)$?
5. Consider the three clusters from Table 15.8. Which pair of clusters will be merged by the hierarchical aggregation technique?
6. An example to be presented to a self-organizing feature map, SOFM, is described by the following attribute vector: $\mathbf{s} = (3, 4, 2)$. Normalize this example to unit length.
7. Explain the difference between normalizing an attribute vector to unit length and normalizing the individual attributes to the unit interval [0, 1].

15.11.2 Give it Some Thought

1. At the beginning of this chapter, we listed some benefits of cluster analysis. Among these was the possibility to identify neurons in RBF networks with clusters instead of with examples. In the case of *k-means*, this is straightforward: each Gaussian center becomes one cluster's centroid. Can RBF networks benefit also from clusters obtained by hierarchical aggregation?
2. Develop a machine-learning approach that first pre-processes the training examples using a cluster analysis technique, and then uses them for classification purposes.
3. Explain how self-organizing feature maps can be used to define the code vectors to initialize *k-means*. Will it make sense to consider the opposite approach (initialize SOFM) by *kmeans*)?
4. The idea of using *auto-encoding* for noise removal is somewhat controversial. Under what circumstances do you think it will succeed and when do you think it will fail?

15.11.3 Computer Assignments

1. Write a program that accepts as input a training set of unlabeled examples, chooses among them k random code vectors, and creates the clusters using *k-means*.
2. Write a program that decides whether a pair of clusters (obtained by *k-means*) should be merged. The easiest way of doing so is to compare the distance between the two clusters with the average cluster-to-cluster distance in the given set of clusters.
3. Write a program that creates the clusters using the *hierarchical aggregation* technique described in Sect. 15.4. Do not forget that the distance between clusters is evaluated differently from the mechanism used by *k-means*.

4. Write a program that accepts a set of unlabeled examples and subjects them to the training of a self-organizing feature map (SOFM).
5. Write a program that uses a SOFM (trained in the previous assignment) for data visualization.
6. Implement a single-hidden-layer MLP and use it as an *auto-encoding* network for feature-reduction purposes. Then run a decision tree generator separately on the original training set, and then on the same examples re-described by the new features. Does the second decision tree outperform the first?
7. Take a training set without class labels. Remove 10% of the examples. Train an *auto-encoding* network on the remaining 90% examples until an almost perfect 1-to-1 mapping is achieved. In the examples that have been removed from the training set, hide some of the attribute values (or replace these values with zeros). Present these examples to the trained AE-network and see if the network outputs reasonably accurate estimates of the hidden values. Experiment with different percentages of the hidden values and decide how many attributes must be present for the AE-network to fill-in the rest.

Chapter 16
Deep Learning

Traditional machine learning has its limitations. For one thing, the nature of the class we want to learn can be so complicated that simple techniques fail. For another, the excessive detail of available attributes may obscure vital information about the data. To cope with these complications, more advanced techniques are needed. This is why *deep learning* was born.

Deep learning relies on neural networks that are now implemented in ways that make the good old MLP seem obsolete. Different activation functions and different ways of quantifying error are used. The number of hidden layers has grown significantly. And advanced data-transformation operators, borrowed from the field of computer vision, are in common use.

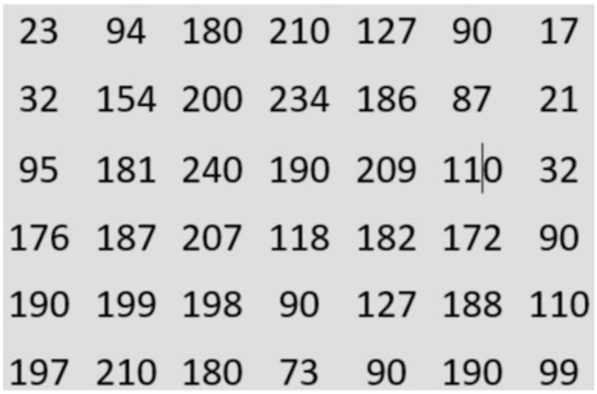
The chapter describes all these novelties and explains how they help us overcome earlier obstacles. The text then proceeds to computational costs, the need to create synthetic data, and *transfer learning*. Importantly, *deep learning* is presented as just another approach to machine learning, an approach that is subject to the same theoretical constraints and practical challenges that the previous chapters discussed in such detail.

16.1 Digital Image: Many Low-Level Attributes

Perhaps the first major beneficiary of deep learning is the field of computer vision; and while this is not its only application domain, it is instructive enough to serve as a useful framework for the explanation of certain basic principles.

Digital Image and Pixels Digital images consist of picture elements called *pixels*. A pixel is the smallest discernible unit, essentially a point that, in a black-and-white picture, is represented by an integer from the one-byte interval, [0, 255]. The concrete value quantifies the pixel's shade of gray, where 0 means totally white and

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_16



23	94	180	210	127	90	17
32	154	200	234	186	87	21
95	181	240	190	209	110	32
176	187	207	118	182	172	90
190	199	198	90	127	188	110
197	210	180	73	90	190	99

Fig. 16.1 A small segment of a black-and-white digital image: a matrix of integers, each representing one degree of gray

255 totally black or the other way round. The entire image is a very large matrix (easily 1000 × 1000) of these integers. Figure 16.1 shows a small segment.

Leaving the realm of black-and-white images, each color is a composition of red, blue, and green. The color image is thus described by three times as many pixels. This causes two principal problems: attribute vectors are impractically long, and the level of detail they provide is such as to render individual attributes almost meaningless.

Edges and Objects The discipline of computer vision has developed countless algorithms to make sense of the raw data. Some of these algorithms detect *edges*. A closer look at the picture detects areas marked by small values (light gray), and areas with higher values (dark gray). An *edge* is the boundary between the light area and the dark area. Thus in the first line in Fig. 16.1, an edge is between 94 and 180.

Computer vision knows how to combine edges into lines, then squares, circles, cubes, all the way up to such objects as, say, a family house. The idea is to start with lower-level features, to process them (e.g., by removing noise), and to build from them simple objects, then more advanced objects, finally obtaining something that makes sense of the picture's contents.

From Texture to Segmentation There is more to an image than just edges. A human observer easily distinguishes metal from leather by their textures; image-processing techniques do something similar. Quite a few techniques can be used here. Some are straightforward, such as the statistical features that quantify average light intensity or its variations. Others, such as the so-called *Gabor wavelets*, are fairly sophisticated. From our perspective, statistical features and wavelets play the role of example-describing attributes.

The contents of a given region in an image are marked by specific values of the texture features. These change from one region to another, which can be important in medical applications: one area of an MR image can represent healthy brain tissue; another, a tumor. Each has its own texture, and computer vision discovers these regions by a process called *segmentation*. The result can be the information about the tumor's size and shape.

Machine Learning and Vision: Traditional Approach Historically, the oldest attempts to apply machine learning to computer vision assumed that visual objects would first be redescribed by attributes such as edges, objects, or textures. From classified examples thus described, machine-learning software would induce the requisite classifiers.

The results were often unconvincing, either because the attributes failed to characterize the images adequately, or perhaps because there were too many of them. Many computer-vision specialists became skeptical.

***Deep Learning*'s Approach** The situation changed when scientists realized that too much work had been left to computer vision. After all, machine learning can do more than just induce classifiers. It can select the best attributes, it can even create useful higher-level features. For instance, this is accomplished by the hidden layers of multilayer perceptrons and RBF networks.

Figure 16.2 depicts the change in the mindset that has led to deep learning. In classical machine learning, the attributes are chosen by the engineer. By contrast, deep learning creates useful features automatically from such low-level attributes as pixels in digital images.

Attributes and Features A terminological comment is in place, here. Some fields—such as computer vision, neural networks, and statistics—prefer the term *feature* instead of *attribute*. For the sake of continuity with earlier work in these disciplines, this section will therefore use the terms *attributes* and *features* interchangeably.

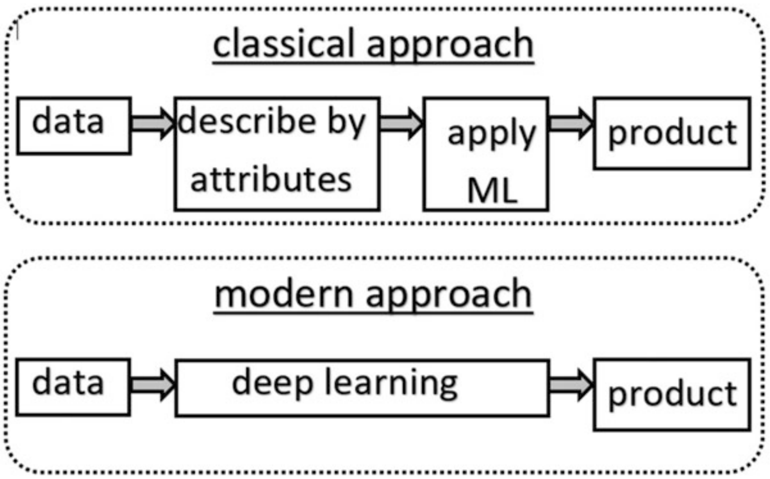


Fig. 16.2 Deep learning not only learns to classify but also discovers useful higher-level features with which to describe the examples

16.1.1 *What Have You Learned?*

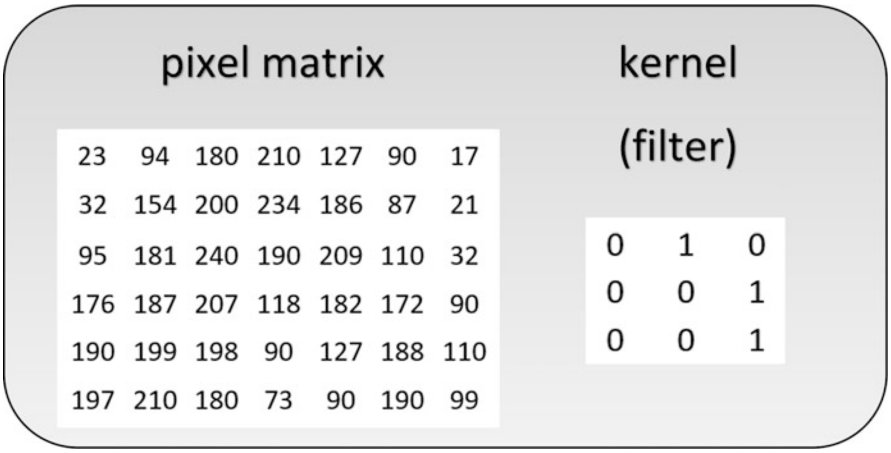
To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain how the digital image consists of pixels. How does the field of computer vision identify edges and objects? What is *texture* and how is it used in segmentation?
- Briefly characterize the oldest attempts to apply machine learning to images characterized by edges, objects, and textures.
- What is the main "trick" employed by deep learning? Why does deep learning so soften outperform classical machine learning?

16.2 Convolution

Let us acquaint ourselves with the operations at the core of the deep learning paradigm. First of them, *convolution*, has lent the name to the most popular approach: *convolution neural networks*.

Kernel*, also Known as *Filter On the left-hand side of Fig. 16.3 is the same excerpt from a pixel matrix that we met in the previous section. On the right-hand side is a 3-by3 matrix that we will interchangeably call *kernel* or *filter*. The kernel represents a specific pattern, in this case is a vertical line slightly turned to the left at the top. Convolution is to determine whether the pattern represented by the kernel can be


pixel matrix

23	94	180	210	127	90	17
32	154	200	234	186	87	21
95	181	240	190	209	110	32
176	187	207	118	182	172	90
190	199	198	90	127	188	110
197	210	180	73	90	190	99

kernel

(filter)

0	1	0
0	0	1
0	0	1

Fig. 16.3 The matrix on the left is an excerpt from an image. The matrix on the right is a *kernel* (also called *filter*)

detected in the image, and if so, where exactly it is located and how strongly it is pronounced.

Suppose the elements in the pixel matrix are denoted by $m_{i,j}$ and the elements in the kernel are denoted by $k_{i,j}$. The degree of fit is defined by the following formula.

$$F_{x,y} = \sum_{i,j} m_{x+i,y+j} \cdot k_{i,j} \tag{16.1}$$

The area in the pixel matrix must have the same dimensions as the kernel.

Numeric Example Consider the upper-left corner of the pixel matrix in Fig. 16.3. We take the kernel shown on the right and place it over the this 3-by-3 segment of the pixel matrix. The degree of fit is calculated as follows (there are only three terms because all other fields in the kernel are 0):

$$F_{0,0} = 94 \cdot 1 + 200 \cdot 1 + 240 \cdot 1 = 534$$

Let us now shift the same kernel over the pixel matrix one pixel to the right. Here is the new degree of fit:

$$F_{0,1} = 180 \cdot 1 + 234 \cdot 1 + 190 \cdot 1 = 604$$

We observe that the degree of fit is here somewhat higher than in the previous case. This means that, as we moved to the right, the kernel's pattern strengthened.

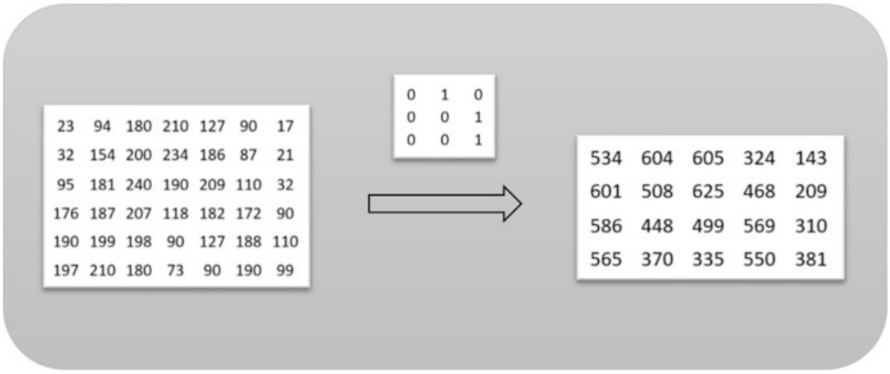
Some Comments The kernel's degree of fit tells us how strongly (or weakly) the kernel's pattern is pronounced in different locations in the image. Of course, the kernel from Fig. 16.3 was very simple: it only contained zeros and ones to make the calculations easy. In reality, any numbers can be used, not just integers, and they can be positive as well as negative. As a result, virtually any curve or specific shape or pattern can be represented by some kernel.

Different kernel sizes are often used, such as 2×2 or 5×5, though rarely larger than that. Most common are square kernels, but rectangles are sometimes used, too.

Where do the Kernels Come from? While it is possible to create the kernels manually, they are usually induced from data. The induction principles are reminiscent of the backpropagation of error known from multilayer perceptrons, but certain aspects make convolution neural nets somewhat different from the classical approach.

The entire Sect. 16.4 will be devoted to the question of kernel induction. Before we reach that stage, however, we need to get acquainted with some details that, while not immediately obvious, have to be understood before we can start writing the computer program.

Sliding Window To discover the presence of the given pattern, the kernel is applied systematically by a *sliding window*. The principle is illustrated by Fig. 16.4 where



23	94	180	210	127	90	17
32	154	200	234	186	87	21
95	181	240	190	209	110	32
176	187	207	118	182	172	90
190	199	198	90	127	188	110
197	210	180	73	90	190	99

0	1	0
0	0	1
0	0	1

534	604	605	324	143
601	508	625	468	209
586	448	499	569	310
565	370	335	550	381

Fig. 16.4 Sliding the kernel along the pixel matrix on the left results in the (somewhat smaller) matrix on the right

the pixel matrix on the left has been transformed by the given kernel into the matrix on the right.

Here is how the transformation is accomplished. At the beginning, the window containing the kernel is applied to the upper-left corner of the image matrix, obtaining the value 534 which is then entered in the field in the upper-left corner of the matrix on the right. After this, the kernel is shifted by one pixel to the right ($x = x + 1$), and the degree of fit calculated, which results in 604 being entered to the next field in the resulting matrix.

The shifting is then repeated, one pixel at a time, until the limit of the image has been reached. After this, the window returns to the left but is shifted down by one pixel ($y = y + 1$), which means that its upper-left corner now corresponds to the square containing 32. The degree of fit is here 601, which is the number at the coordinates $[1, 0]$ of the resulting matrix. The process is repeated in this manner until the window reaches the bottom-right corner of the image.

Two Tasks The task for the sliding-window mechanism is to calculate the kernel's degree of fit for all possible locations in the image. The higher the degree of fit, the stronger the kernel's pattern's presence.

The second task is data reduction. Whereas the original pixel matrix was 6×7, the resulting matrix is 4×5. This reduction can be repeated by subjecting the resulting matrix to another kernel (in a neural network's next layer), and then to another and another, each time reducing the data size.

Stride The window does not have to shifted by just one pixel. Actually, the size of the sliding step is an important parameter, called *stride*. The stride can be different along the horizontal axis and along the vertical axis. For instance, suppose the engineer has chosen $s_x = 2$ and $s_y = 3$. The window is then always shifted by 2 pixels when moved to the right ($x = x+2$), and by 3 pixels when moved downward ($y = y + 3$).

Figure 16.5 illustrates the case where $s_x = 2$ and $s_y = 2$.

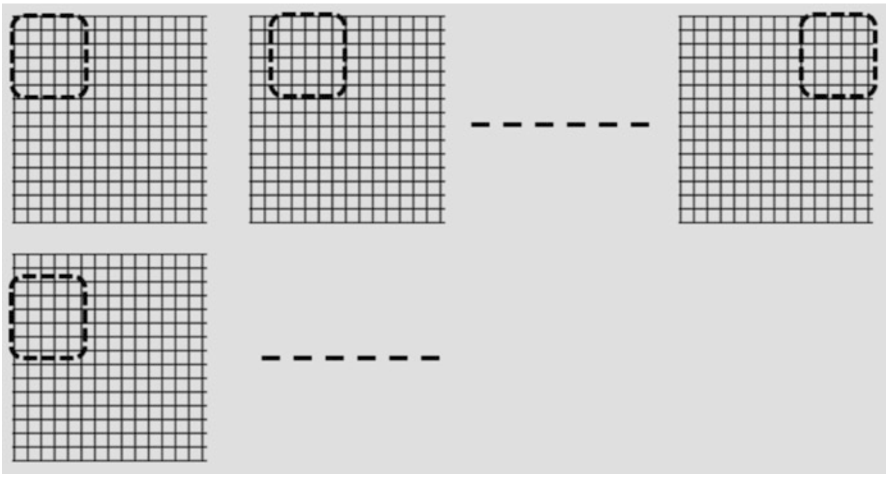


Fig. 16.5 In this picture the window is sliding with strides $s_x = 2$ and $s_y = 2$

With greater stride, the data reduction is more significant. Besides, higher values of s_x and s_y mean smaller computational costs because fewer degrees of fit then have to be calculated. The price to be paid for these savings is a certain loss of detail, which sometimes is affordable—and sometimes not.

Valid Padding* and *Same Padding When describing the principle of the kernel's sliding window, the previous paragraph neglected one important detail: what to do at the end of the row and at the end of the column? Should the sliding be stopped before the kernel reaches outside the image's borders? This would seem natural because how can we calculate the degree of fit, $\sum m_{i,j} \cdot k_{i,j}$, if the values of $m_{i,j}$ outside the matrix do not exist?

Two alternative approaches exist. The first, *valid padding*, does indeed stop the sliding if the next $x = x + s_x$ or $y = y + s_y$ causes the window to reach beyond the image. In the case of larger kernels and higher values of s_x or s_y, this may result in a situation where the last columns or rows of bits are ignored.

The second approach, *same padding*, continues the sliding even beyond the image's border, using for all "pixels" outside the image $m_{i,j} = 0$ (see Fig. 16.6). In this way, no columns and rows are ignored. However, the zeros can be regarded as somewhat artificial.

Whether *valid padding* or *same padding* is to be preferred depends on the concrete application and the engineer's experience and personal preferences.

Not Just One Kernel In reality, there is not just one kernel. Typically, one layer in the convolution neural network will use several of them in parallel, each representing a different pattern.

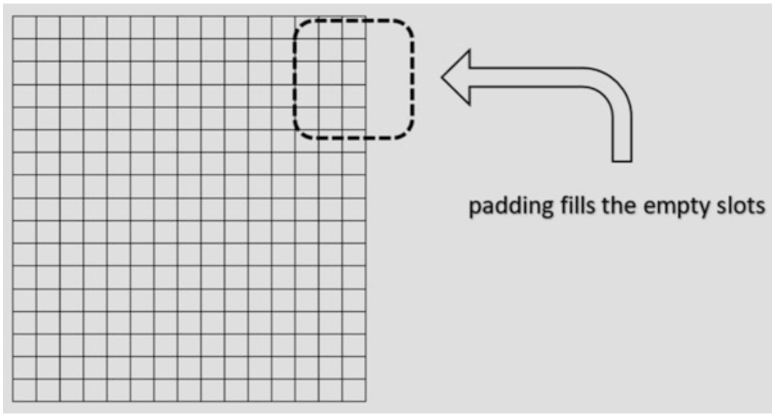


Fig. 16.6 Suppose the kernel's size is 5×5 and the horizontal stride is $s_x = 3$. At the end of the horizontal sliding, the kernel can reach beyond the end of the image. The "missing values" can be filled by *padding*

16.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the kernel (filter) and how do we calculate its degree of fit with a given portion of the image?
- Explain the principle of the sliding window, and define the following terms: stride, valid padding, and same padding.
- How do different values of strides affect data reduction and computational costs? What is the impact of kernel size?

16.3 Pooling, ReLU, and Soft-Max

Now that we understand the principles of convolution, we are ready to take a look at other aspects of *convolution neural networks,* CNNs.

Max-Pooling Pooling is used to further reduce the unnecessarily detailed data. The principle is shown in Fig. 16.7 where each 2-by-2 square is replaced by the highest number it contains. For instance, the square in the upper-left corner contains 534, 604, 601, and 508. The highest among them is 604, and this is therefore the number entered at the location [0, 0] of the matrix on the right.

It is important to understand *why* max-pooling is used. The convolution step calculated the strength with which the kernel's pattern is represented at each location. However, we usually do not need to express the location with the precision

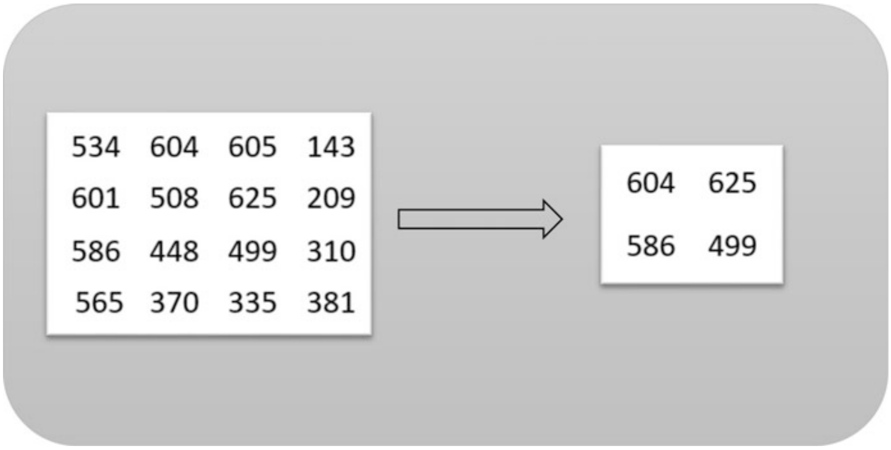


Fig. 16.7 *Max-pooling* replaces each 2×2 square with the greatest of the four numbers it contains. *Ave-pooling* would replace them with their averages

of a single pixel. The maximum number in the 2-by-2 square gives the maximum expression of the pattern of in this square. The 2-by-2 resolution is more than sufficient for many practical purposes.

The ability of max-pooling to reduce the size of the data is obvious. Of course, instead of a square, a rectangle (of any size) can be used. The concrete extent of pooling is determined by the engineer. Here are the trade-offs. The smaller the data, the fewer the system's parameters, and the lower the computational cost. On the other hand, excessive reduction may destroy important details.

Ave-Pooling Instead of the max-pooling from the previous paragraph, *ave-pooling* is sometimes used. The only difference is that instead of using the maximum in the given square, we calculate the average. Thus for the upper-left square in Fig. 16.7, this average is (after rounding to integers) $(534 + 604 + 601 + 508)/4 = 562$. Note that this value does not depend just on the single maximum but is affected by all numbers in the square.

Activation Function Chapter 6 explained that in multilayer perceptrons, the neurons should have non-linear activation functions. For a long time, it was generally accepted that the `sigmoid` and similar functions are the only reasonable choice, made attractive by the elegant first derivative (which led to simple backpropagation formulas).

Later, however, experience revealed that a much simpler non-linearity can be used without any detriment to classification and learning. Particularly popular are currently the two alternatives shown in Fig. 16.8. We have already met them toward the end of Chap. 6. Their advantages are obvious. First, the output values are no longer limited to a fixed interval such as (0, 1). Second, the first derivatives are even more easily obtained. For the constant segment in ReLU, the first derivative is 0;

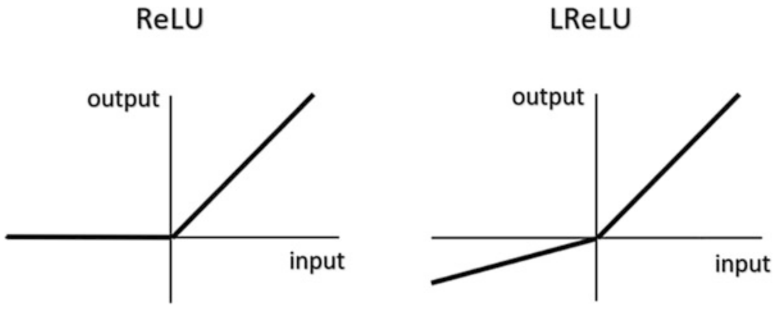


Fig. 16.8 Two activation functions commonly used in CNN: *Rectified linear unit* (ReLU), shown on the left, and the *leaky ReLU* (LReLU) shown on the right

and for the linearly growing segment it is 1. The derivatives of LReLU are just as simple. This will come handy in the backpropagation of error for training.

Probabilities Obtained by *Soft-Max* For the activation function,[1] the neurons in the classical MLPs typically relied on `sigmoid` and `tanh` functions. The former limited the output to the interval (0, 1); the latter, to $(-1, 1)$. Convolution neural networks prefer to subject the output to the *soft-max* function (Chap. 6) which we know converts the output values to probabilities by making the numbers in the vector sum to 1. The reader will recall the formula where y_i denotes the original output and p_i the corresponding probability:

$$p_i = \frac{e^{y_i}}{\Sigma_j e^{y_j}} \tag{16.2}$$

Normalization In the digitized image, all pixel values come from the interval $[0, 255]$. For the needs of machine learning, however, it is usually recommended that they be normalized either to [0, 1] or to $[-1, 1]$. In that case, application of the kernel will not so easily lead to ever-growing numbers. For normalization into [0, 1], it is enough to divide all values by 255. For normalization into $[-1, 1]$, we do the same and then multiply by 2 and subtract 1.

There is another motivation for normalization. One of the consequences of abandoning `sigmoid` in favor of the simpler ReLU is that the output, y_i (which in `sigmoid` was constrained to [0, 1]) can now be a very big number, so big that calculation of e^{y_i} may cause overflow.

Numeric Example Suppose that our neural network has three outputs whose values for a given example are 0.3, 0.8, and 0.2, respectively. The corresponding *soft-max* values are given in the following table.

[1] Recall that this is sometimes called *transfer function*.

Neural output, y_i	0.3	0.8	0.2
Soft-max output, p_i	0.28	0.46	0.26

Note that the values resulting from *soft-max* sum to 1 so that their interpretation as probabilities does not violate the postulates of the theory of probability.

Network's Architecture Figure 16.9 gives us an idea of a typical architecture of a convolutional neural network. The basic building block is the convolution layer (typically consisting of a set of different kernels) followed by pooling. The task for convolution is to discover specific patterns; pooling reduces the data size. Recall that the size of the data is also reduced by convolution if the strides are greater than 1.

At the top of the network the engineer often adds a multilayer perceptron, usually with the ReLU or LReLU activation functions instead of the traditional `sigmoid`. MLP's output is then subjected to *soft-max*, and the output of *soft-max* is used to calculate the network's *loss* for the given example as explained in Sect. 6.6.

A little revision will do no harm. Suppose the correct class for the given example is C_i and let the i-th output (after *soft-max*) be p_i. CNN's loss is then calculated as follows:

$$L = -\log_2 p_i \tag{16.3}$$

CNNs are Deep The reader will recall that various reasons prevented classical MLPs from using more than two or three hidden layers. To begin with, *vanishing gradients* render additional layers ineffective; besides, too many layers mean too

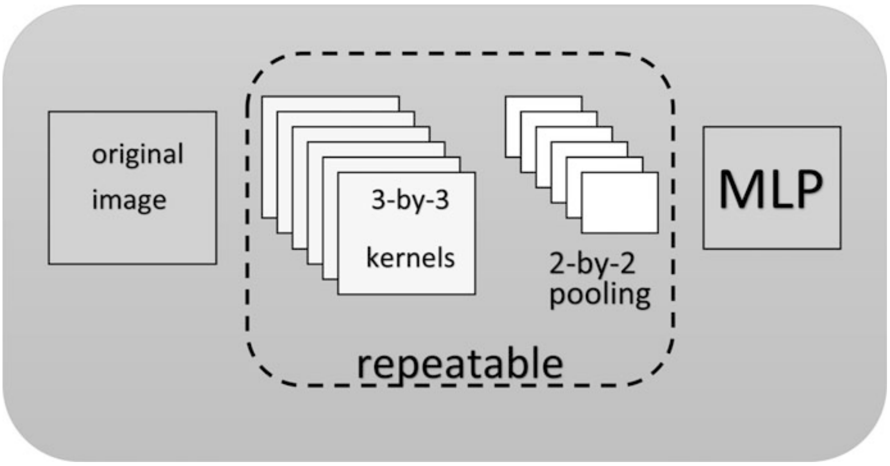


Fig. 16.9 A simple CNN architecture: layers of kernels are followed by pooling; after several repetitions, the resulting values are fed into a MLP-network, whose outputs are then subjected to the soft-max function

many trainable parameters, and these can render training by backpropagation prohibitively expensive computationally.

When the neural activation functions are ReLU or LReLU, the problem of vanishing gradients all but disappears (because neural outputs are not bounded by the `sigmoid`'s $(0, 1)$-interval). As for the computational costs, these are often deemed acceptable in view of the power of today's computers, especially if they are equipped with *graphical processing units, GPUs,* that facilitate massive parallelization of the training algorithm.[2]

The designer of a CNN is thus much less constrained than in the past, and the number of layers can be quite high. The pair convolution/pooling is usually repeated several times, sometimes many times, so that the network can easily consist of dozens of layers. In this sense, the networks are *deep*, a circumstance from which this paradigm derives its name: *deep neural networks* or *deep learning*.

Boundless Freedom What makes CNNs particularly attractive is their unparalleled flexibility. A cursory glance at Fig. 16.9 will help you develop an idea. To begin with, the kernels' dimensions do not have to be 3×3 as in the picture; the kernels can be small or large, square or rectangular (or even of other shapes, though those are rare). Various values for horizontal and vertical *strides* can be employed. Then there is the choice between *valid padding* and *same padding*. And instead of the 2×2 squares for *max-pooling*, larger ones can be used. Alternatively, *max-pooling* can be replaced with *ave-pooling*.

The engineer also has to decide how many convolution/pooling pairs are to be used. If the task is easy, one pair may be sufficient. If the pattern recognition problem is extremely hard, a great many layers may be necessary. Each layer can have a different type of kernels and a different number of kernels. Sometimes, the *pooling* step is skipped, and two or more convolution layers immediately follow each other.

The last source of flexibility is provided by the multilayer perceptron placed at the network's top. It can have one or two or three hidden layers, each with its own size, the engineer can consider diverse activation functions, and so on. And of course, the engineer may decide not to use MLP at all.

Deep Learning: More Art Than Science So many parameters, so many degrees of freedom, so many opportunities for changes, modifications, adjustments. CNN does not know ready-made recipes, at least not many of them. More than any other machine-learning paradigm, CNN is more art than science. Most of the time, the engineer depends on his or her own experience, creativity, and imagination. Rather than on handbooks and manuals, success depends on systematic experimentation, statistical evaluation of alternatives, and common-sense considerations.

[2]Today's PCs with GPU's are easily by five orders of magnitude faster than the workstations available to the pioneers of neural networks in the late 1980s. Calculations that now take one second then took a whole day ... and the computer froze at least once a week, sometimes several times a day.

16.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of m*max-padding* and *ave-padding*, and discuss their role in a convolution neural network.
- Explain the principle of the *soft-max* output. Why do we use it?
- What are the two most popular transfer (activation) functions used in CNN's neurons? Why could the more sophisticated `sigmoid` be abandoned?
- What are the most important aspects of a concrete CNN's architecture? What constitutes its remarkable flexibility?

16.4 Induction of CNNs

Let us now turn to the main question: how can a convolution neural network be induced from data? The principle is the same as in the multilayer perceptrons from Chap. 6: backpropagation of error. This said, there are few differences that we need to discuss.

Linear Classifiers Return to Fig. 16.4 and recall how the values in the matrix on the right were calculated: they were obtained by element-by-element multiplication of the image matrix by the kernel matrix. More specifically, the following formula was used:

$$F_{x,y} = \sum_{i,j} m_{x+i,y+j} \cdot k_{i,j} \tag{16.4}$$

Note that each field in the resulting matrix is a linear function of the corresponding window in the pixel matrix. The weights are the kernel's elements! Practically this means that each kernel can be treated as a neuron of sorts. For its transfer function, either ReLU or LReLU is typically used. The reader will recall that perceptron learning, as well as the neurons in a MLP, benefited from a *bias*, an additional, "zeroth" weight that serves as a coefficient of an artificial input whose value is always 1.

Trainable Parameters The weights to be trained in CNN are therefore the values of the kernels to which possibly the zeroth weights (serving as biases) are added. Each location of the sliding window then represents a training example. By contrast, pooling does not involve training.

In those CNN architectures that are topped by an MLP (Fig. 16.9), we also have to train its hidden-layer weights with the backpropagation of error we already know from Chap. 6. The only difference is in the transfer function: `sigmod` or `tanh` are rarely employed, here.

Loss Function Let us revise how to calculate the error to be backpropagated. Suppose the network's outputs have been subjected to *soft-max*, resulting in probabilities, and suppose that a training example **x** belonging to the i-th class has been presented. Once the example has been forward-propagated through the network, the observed probability returned by the network for this i-th class is p_i. The error for this example is then quantified by the *loss* calculated as follows:

$$L = -\log_2 p_i \tag{16.5}$$

The higher the value of p_i (the probability assigned by the network to **x**'s known class), the more correct the network is. Conversely, small values of p_i indicate that something is wrong, and weights-modification is called for. This is indeed what Formula 16.5 reflects. For instance, if $p_i = 0.9$, then $L = 0.15$, and if $p_1 = 0.3$, then $L = 1.74$. We can see that the smaller the probability, the greater the loss to be backpropagated. This is similar to what we got used to in multilayer perceptrons where greater error meant stronger weight adjustment. The only difference is that, in CNN, the role of the mean squared error, MSE, has been taken over by the loss function, L.

Loss Function Is Related to Information Theory The reader will recall this formula's relation to the information contents of the message, "**x** belongs to the i-th class" (see Chap. 5). Information theory calculates the information contents by base-2 logarithm, but one can just as well use natural logarithm, ln(), because $\log_2 x = \frac{\ln x}{\ln 2}$: this means that the two logarithms differ only by the factor $\ln 2$, which is a constant.

Backpropagation Just as in the case of MLPs, the output of a convolution neural network is simply a function of its inputs. This function depends on a great many parameters, and the task for backpropagation learning is to modify the values of these parameters in a way that minimizes the network's error—which in the case of CNN is measured by *loss*, $L = \ln p_i$, where p_i is the probability assigned by the network to the known class of the training example:

$$p_i = \frac{e^{y_i}}{\Sigma_j e^{y_j}}$$

Here, y_i is the network's i-th output, which is a function of the inputs and of the weights (kernels and biases). If we want to find the most beneficial change in a certain weight, w, calculus recommends to find the gradient which, following the chain rule of differentiation, leads to the following:

$$p_i = \frac{\partial L}{\partial w} = \frac{\partial L}{\partial p_i}\frac{\partial p_i}{\partial w} = \frac{\partial L}{\partial p_i}\frac{\partial p_i}{\partial y_i}\frac{\partial y_i}{\partial w}\dots \tag{16.6}$$

The procedure is repeated for every single trainable parameter. Even if laborious, the calculation is straightforward. First derivative of $\ln x$ is known to be $1/x$. Somewhat more challenging is the first derivative of the exponential function for p_i, but the others are easy. We just have to keep in mind that ReLU has to be differentiated piece-wise, separately for the positive part (where the first derivative is a constant) and the negative part (where the first derivative is zero). The piece-wise first derivative of LReLU is also simple.

Initialization Before the parameters (weights) can be trained, they have to be initialized. In principle, they should start with small values—but how small is *small*? In multilayer perceptrons, the idea was to make sure that the sum, $\sum_i w_i x_i$, is somewhere in the linear region of the `sigmoid` function. In CNN, different transfer functions are used, but the same mechanism for weight initialization is widely used: *Xavier initialization*.

Suppose that a given neuron has N_{in} inputs and N_{out} outputs. Weights are initialized by a random-number generator that chooses the values from a Gaussian (normal) distribution whose mean is zero, and standard deviation set by to the following formula:

$$\sigma = \sqrt{\frac{2}{N_{in} + N_{out}}} \tag{16.7}$$

No Need to do All the Programming Fortunately, the programmer does not have to worry about the numerous details of backpropagation because today's programming languages can do the differentiation automatically. This is the case of Matlab, Python, or various higher-level packages such as Google's Tensorflow or PyTorch. The programmer only has to specify the kernels, activation functions, and the other parameters that characterize the given CNN.

Engineer's Decisions Instead of risking a bug in the programming of backpropagation, the engineer can focus on the many practical decisions to be made, each of them having the potential to improve (or worsen) the CNN's chances of success: padding, strides, number of layers, dimensions of kernels, the size of the MLP on CNN's top, the choice of transfer functions (ReLU and LReLU), and some specific details required by the concrete software.

Besides, there are many decisions related to the organization of the training and testing experiments, such as whether the examples should be presented one at a time or in batches, whether random sub-sampling of N-cross-validation is to be used, how to go about statistical evaluations of the results, and so on.

16.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the *loss* function and how is its value calculated? How does it relate to the *soft-max* output?
- What is *Xavier* weight initialization? What is its purpose?
- Explain the principle of the backpropagation training in CNN. In which aspects is it different from the training of MLPs?
- Which CNN parameters are trained by the backpropagation mechanism?

16.5 Advanced Issues

Now that we understand the fundamentals, we are ready to proceed to certain advanced concepts characteristic of modern practice of CNNs.

Examples of More Than Two Dimensions For simplicity, we have up till now considered only black-and-white images where each pixel is represented by a single integer. In color images, however, each pixel is described by a vector of three *channels*—integers that quantify the light intensity of red, blue, and green (RBG). Each image is thus described by *three dimensions*. The engineer has to decide whether to use the same kernel for each channel.

Moreover, it is a common practice to present the training examples not one at a time, but in batches that consist of N_{batch} examples. In this event, the input to the CNN-training program has *four dimensions* (in the case of color images).

Beyond Images In *video* applications, each example is a series of images, and this adds yet another dimension. A single training example then has four dimensions, three of them specifying light intensities in the individual channels for a single frame, and the last for the sequence of frames. If these examples are presented in batches, the input data have *five dimensions*, something rather rare in programming outside the realm of machine learning.

By contrast, *audio* can be represented by a sequence of numbers, and an example can then be represented by a vector. The kernels are then one-dimensional, too.

Tensors Previous paragraphs convinced us that we need CNNs to be capable of working with diverse numbers of dimensions. For domains of this kind, mathematicians have developed a generalized data structure known as *tensor*. A tensor can have any number of dimensions, and the dimensionality is defined by *shape*. For instance, a vector of length 4 is a tensor of shape [4], a 3-by-4 matrix is a tensor of shape [3, 4], a 3-by-5-by-4 matrix is a tensor of shape [3, 5, 4]. A scalar is a tensor whose shape is *null*.

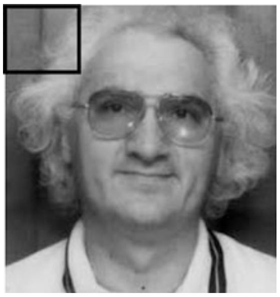


Fig. 16.10 Looking for a concrete object, say a `nose`, a sliding window is passed along the image. The sliding window represents a CNN capable of recognizing the presence of the given object

The reason we are mentioning tensors, in this context, is that advanced programming tools, such as TensorFlow, have built-in functions capable of operating over tensors instead of over classical arrays. All CNN-related programming is thus greatly simplified. For instance, if the engineer decides to work with three dimensions instead of two, essentially the same code can still be used with only minor modifications.

Recognition of Simple Objects One typical task can be illustrated by the questions, "is there a soccer ball, in this image?" and "how many birds are there?" In other words, the software is to look for relatively simple objects. In this case, it is not necessary to train the network on entire images; it is enough if the input field is something like, say, 100-by-100 pixels, which significantly reduces the number of trainable parameters.

Once the CNN has been trained, it is applied to future images by way of a "sliding window." Figure 16.4 illustrates the principle. A network that has been trained to recognize, say, a `nose` is applied first to the upper-left corner of the image and is then moved, step by step, a few pixels to the right or (after the completion of the horizontal) a few pixels down. At each step, the CNN returns a 1 if the 100-by-100 window contains a nose, and 0 otherwise.

Usually some post-processing is then needed. Once the classifier has identified the object, it is more than likely that the same object will still be in the window after the shift of a few pixels. Some master algorithms have to decide whether this is still the same nose or another nose in its close neighborhood (Fig. 16.10).

Many Trainable Parameters: Large Training Set Is Needed Machine learning knows that the more trainable parameter there are, the more training examples are needed. This, after all, follows from the computational learning theory introduced in Chap. 7. In the case of CNN, this means that the number of examples should exceed the number of weights. This can become a serious problem in many applications. For very difficult recognition patterns, many neural layers are necessary, and the network can easily end up with millions of weights.

In computer vision, a training set consisting of millions of examples can sometimes be obtained. For a start, Google search can often find a large initial pool. Additional training examples can then be created from those in the initial pool by adding noise, cropping, rotating, zooming in and zooming out, and other such techniques.

In other fields, however, training examples may be expensive, or simply not available at all. Thus in a medical domain, a training example may consist of the data describing a concrete surgery. The number of surgeries being limited, the training set can never be very large.

Computational Costs Even if we *do* have a sufficiently large training set, the computational costs of CNN training may render the whole process impractical.

Suppose our CNN has 10^5 trainable weights. To prevent overfitting, at least 10^6 examples are needed. This means that, in a single epoch, $10^5 \times 10^6 = 10^{11}$ weights have to be modified. If the task is really hard, a fairly modest expectation will suggest that the training will not converge before, say, 10^4 epochs. This means the total of $10^{11} \times 10^4 = 10^{15}$ weight updates. Even if the computer is capable of accomplishing 10^9 updates in a second, the learning process will take $10^{(15-9)} = 10^6$ s, which can be something like 2 weeks.

If we increase the number of weights ten times, and the number of epochs also ten times, the training will take years.

Transfer Learning A popular way of dealing with unrealistic computing costs is known as *transfer learning*. The idea is to take advantage of an existing CNN that has been trained on a similar domain, and to modify it to our specific needs.

Suppose that someone has induced a CNN—let us denote it by C_0—that has proved successful on a specific computer-vision task. Facing the need to induce another CNN for a similar task, the programmer does not have to start from scratch, from randomly initialized weights. Rather, she may prefer to start with C_0 to whose top she just adds a single-hidden-layer MLP. The new training set is then employed only for the training of the weights of this added MLP, while the weights in the original C_0 are left unchanged (we say, they are "frozen").

The reader understands why this works. The training of the original network C_0 has created meaningful higher-level features that may come handy even in the new task; in other words, they can be *transferred* to this new task. Since only the highest-level weights are trained, the number of trainable parameters is reasonable. Consequently, smaller training set will suffice, and also the computational costs will be manageable.

16.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the usual number of dimensions of the training data? What are *tensors*? Why are they so popular, in deep learning?
- What are the consequences of the very high number of trainable parameters in deep learning?
- Explain the principle of *transfer learning*. Under what circumstance do we use it and under what circumstances is it likely to succeed?

16.6 CNN Is Just Another ML Paradigm

Amazed at CNN's power, some experts tend to neglect lessons and experiences of classical machine learning. This is not reasonable. Deep learning is just another machine-learning paradigm. True, it has been spectacularly successful. But at the end of the day, it has to be evaluated by the same performance criteria, using the same statistical techniques, and following the same learning-curve considerations.

Importantly, CNN is prone to suffer from pretty much the same pitfalls as its predecessors. Let us remind ourselves of some of the most critical.

Imbalanced Classes Similarly as the classical approaches, deep learning is sensitive to the problem of *imbalanced classes*: the induced classifier tends to favor those classes that dominate the training set. Fortunately, we already know the mechanisms reducing the negative effects of this tendency. To be more specific, Chap. 11 discussed minority-class oversampling, majority-class undersampling, and classifier modification. All of these techniques are just as relevant in convolutional neural networks.

Performance Criteria Let us not forget that error rate may be a poor indicator of performance. Quite often, the cost of a false negative is different from the cost of a false positive; they may not even be meaningfully comparable. The engineer should consult with the end-users the question of which performance aspect they believe to be important. It can be precision, recall, sensitivity, specificity, or various combinations thereof (Chap. 12). Specific circumstances may even force the users to switch from one criterion for another, just as it happened in the oil-spill domain from Sect. 8.3.

Context and Time-Varying Classes In many applications, the classes we want to learn change their nature with a changed context, just as English phonemes sound a bit different in the UK than in North America. Practical experience has taught us that it is a good strategy to induce a separate classifier for each context, and then use the one that appears to perform better on the given data.

Also, do not forget that the context may change (or evolve) with time. For instance, the nature of certain objects in visual images may be affected by season. Context dependency, as well as the time-varying aspects of some classes, was discussed in Chap. 11.

Multi-Label and Hierarchical Classification In computer vision, it is common that the same example is labeled with two or more classes at the time. For instance, a holiday snapshot can be classified as a beach, blue sky, summer, crowd, flock of birds, and many other things. Sometimes, only some of these classes are important; in other applications, we want to identify as many of them as possible. Besides, the classes may form a sophisticated hierarchy. The engineer has to decide whether plain *binary relevance* is sufficient, or whether something more advanced is needed (Chap. 14).

Insufficiently Large Training Sets Computational learning theory (Chap. 7) teaches us that the more trainable parameters the classifier has, the larger training set is needed. In many domains, however, the examples are expensive, and sometimes they are not available at all. For instance, when planning to apply machine learning to heart transplantation data, the engineer surely cannot expect to have more than hundreds of examples. In this event, deep learning, fashionable though it is, may be a poor choice.

Computational Costs Last paragraph reminded us that high numbers of trainable parameters necessitate very large training sets. This is not without computational consequences. It is not rare that a million weights have to be updated after the presentation of a single training example. If there are a million of these examples, and if a great many epochs are needed, then the training may not be computationally feasible.

Computational costs indeed represent a major obstacle to more widespread use of deep learning. The engineer has to decide whether the just-a-little-bit-higher classification performance merits the weeks of calculations on a supercomputer.

Opportunities Offered by Boosting One way to reduce the computational costs is offered by the boosting algorithms from Chap. 9. The idea is to induce a large number of simple classifiers from relatively small subsets of training examples.

16.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How would you deal with the problem of imbalanced classes when implementing deep learning software? How would you handle the problem of multi-label learning?
- Discuss the various consequences of the large numbers of trainable parameters in convolutional neural networks: computational costs, the need for large training sets, the chances of boosting algorithms.

16.7 Word of Caution

In modern world, popularity is a double-edged sword. Once a few spectacular success stories have been reported, the concerned technology acquires the reputation of a cure-all that overshadows everything that existed previously. This is unsound. This section offers a few arguments against such rash attitudes.

Seek the Most Appropriate Tool Any tool's power has to be commensurate with the difficulty of the task at hand; the engineer should always seek the most

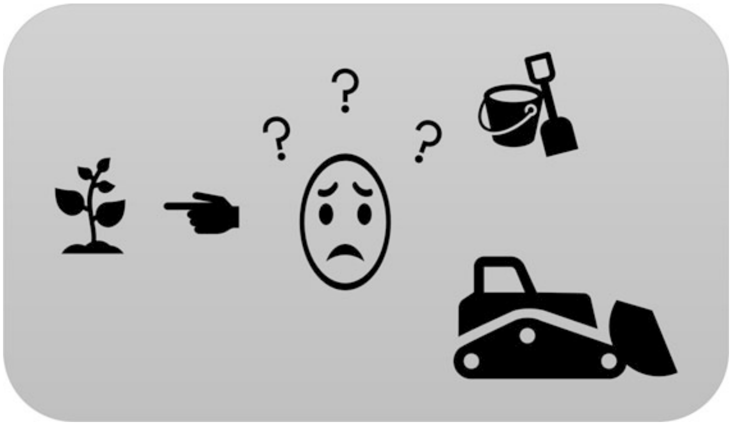


Fig. 16.11 Which tool will remove the weed more efficiently?

appropriate tool. Before reaching for a fashionable new paradigm, we must be sure it indeed offers the best way to handle the job. Popularity is not a technological category; even demonstrated power is not enough.

Figure 16.11 conveys the message by an easy-to-remember metaphor. No matter how powerful the bulldozer is reported to be, the spade will still be the gardener's tool of choice: it does not leave so much mess and is generally handier for the job, even faster (you do not have to wait for the big machine to arrive).

Experience with Hand-Written Characters If you google the term "MNIST" you will reach a website with data files that are now very popular as test-bed in CNN experiments: hand-written digits.[3] The same website provides a table that summarizes the performances scored on these data by diverse machine-learning approaches. Looking at the table, you will realize that the error rate achieved by a linear classifier was 12% whereas the error rates of simple versions of CNN were around 1%, while a really sophisticated implementations of CNN with all sorts of high-level gimmicks did even better, the record being 0.23% (as of Spring 2021). However, the same table also informs us that some k-NN classifiers were not much worse than 1%.

Does this mean that k-NN is about as powerful as CNN? Far from it! What the experience really teaches us is that the domain of hand-written digits is not difficult enough to require the bulldozer from the previous paragraph. Sometimes, reducing the error rate below 1% is critically important. Very often, however, a tiny improvement does not make much of a difference, and the error rate can perhaps be achieved by some simple post-processing.

[3] http://yann.lecun.com/exdb/mnist/.

Of course, we can imagine something much more difficult than character recognition, a task where classical machine learning will fail miserably. This is when deep learning will show its power—provided that we have a sufficiently large training set that adequately represents the given problem.

Programmer Loses Contact with Details The first generation of neural-networks engineers had to implement every possible detail of the learning algorithm in a general-purpose programming language. Those days are over. Nowadays, such languages and open-source packages as Python, Tensorflow, PyTorch, or even Matlab come with rich libraries of built-in functions that make it possible to implement a CNN in a matter of hours.

The consequence is not only higher productivity but also significantly reduced danger of stupid errors. The downside is that the programmer loses contact with the many details of the code. His or her understanding of the behavior of the concrete neural network may thus be negatively impacted.

16.7.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Do we always have to choose the most powerful paradigm? What is the lesson from the handwritten-digits domain?
- In what sense can the availability of powerful programming tools be counterproductive?

16.8 Summary and Historical Remarks

- In such domains as computer vision, the extremely high numbers of low-level attributes render classical machine-learning almost useless. However, sensible information can be induced from higher-level features. One way to create these high-level features is by *convolution*.
- Convolution systematically slides pattern-carrying *kernels* over the matrix of image pixels. Kernels, also known as *filters*, usually take the form of small squares or rectangles. For each location of the kernel over the pixel matrix, the *degree of fit is calculated*.
- Suppose the elements in the pixel matrix are denoted by m_{ij} and the elements in the kernel are denoted by k_{ij}. For the location defined by (x, y), the degree of fit is calculated as follows:

$$F_{x,y} = \sum_{i,j} m_{x+i,y+j} \cdot k_{i,j}$$

- Two parameters, known as *strides* (horizontal and vertical), determine the size of the steps by the kernel slides along the matrix of image pixels. The greater the values of strides, the greater the data reduction.
- Another parameter, *padding*, determines what happens when the sliding kernel reaches the end of the image. Two commonly used alternatives are *valid padding* and *same padding*,
- Another method of data reduction is *pooling*. Most common is *max-pooling*, but the engineer can also consider *ave-pooling*.
- Trainable parameters include the values in the kernels and the weights of the MLP that is sometimes attached at the top of the CNN. Pooling operators do not need training.
- Instead of the `sigmoid` or **tanh** that were used as activation functions in classical neural networks, CNN prefers the simpler ReLU and LReLU functions.
- The topmost layer of a CNN applies the *soft-max* function whose task is to turn the network's outputs to probabilities.
- Instead of the mean squared error, MSE, that is backpropagated in multilayer perceptrons, CNNs backpropagate the value of the *loss* function borrowed from information theory: $L = -\log_2 p_i$ where p_i is the probability assigned by the neural network to the correct class label of the training example, $\mathbf{x}$.
- One typical trait of CNNs is that they can work with data of different dimensionality. Whereas the training examples in classical machine learning are usually described by attribute vectors (1-dimensional data), here they are often 3-dimensional (color images) or 4-dimensional (videos). One extra dimension is added if the examples are presented in batches, rather than one by one.
- The flexibility necessary for working with data of diverse dimensionality is provided by *tensors*—generalized matrices. Their use in modern programming languages greatly simplifies the programmer's job.
- The large number of trainable parameters in CNNs has two consequences. First, the training set has to be very large because the number of training examples has to be greater than the number of trainable parameters. Second, the computational costs of the training can be extreme.
- Computational costs are significantly reduced if the CNN focuses only on a small part of the image, say a 100-by-100 pixel matrix. The induced neural network is then applied to future images by the mechanism of "sliding window."
- Another way to reduce computational costs is by *transfer learning*: a network that has been trained on a different, though similar, application is used as a starting point from which the new training begins. Usually, only the top layers are thus re-trained. Transfer learning usually allows much faster convergence than training the entire CNN, starting with random weights.
- CNN is just another machine-learning paradigm. The engineer has to be prepared to deal with multi-label examples, with context-dependent and time-dependent classes, with the difficulties posed by imbalanced training sets, and so on. Special attention has to be paid to the question of appropriate performance criteria.
- Deep learning offers much more flexibility than traditional machine learning paradigms. The reader already knows, however, that this higher flexibility can

easily be counterproductive. The danger of overfitting has to be mitigated by extremely large training sets. Not only is the training computationally intensive, but the large training set may not even exist.

Historical Remarks In the field of computer vision, kernels and convolution were used as early as in the 1990s, having been introduced by LeCun et al. (1989). The idea of exploiting kernels and convolution in machine learning was first proposed by Krizhevsky et al. (2012) who implemented it in the now-legendary *Alexnet*, perhaps the first known CNN program. In the following years, they kept improving this program until it reached a level that could be shown to outperform human subjects, a real landmark. *Xavier* initialization was introduced by Glorot and Bengio (2010).

This author learned a lot from the textbook by Charniak (2018) which also contains a lot of information about how to implement in TensorFlow other machine-learning paradigms, including reinforcement learning.

16.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

16.9.1 Exercises

1. Suppose the input image is described by a matrix of 28-by-28 pixels. Suppose these 784 features are fed to a single layer of eight 5-by-5 kernels applied with strides $S_x = 2$ and $s_y = 2$ if *valid padding*. What is number of outputs of this layer?
2. Figure 16.7 illustrated the application of *max-pooling*. Using the same data, show what happens if *ave-pooling* is applied.
3. Suppose that, for a given example, $\mathbf{x}$, the CNN's given the following three outputs: $(2, 5, 3)$. These are then converted by *soft-max* to probabilities. Suppose that you know that $\mathbf{x}$ belongs to class C_2, the one for which the network's original output was 3. What is the value of the *loss* to be backpropagated?

16.9.2 Give It Some Thought

1. Consider the conflicting aspects of convolutional neural networks with many layers: high flexibility versus the danger of overfitting. Suppose you are presented

with a training set with a certain size. How will this size affect your decision about the CNN's architecture?
2. In what respect are the convolutional and pooling layers better at creating higher-level features than classical neural networks?
3. Whereas classical approaches to machine learning assume that an example has the form of an attribute vector, deep learning typically describes by a matrix. The chapter has even mentioned in passing that examples can be presented as three-dimensional, or even four-dimensional arrays. How does this multi-dimensional view affect the programmer?

16.9.3 Computer Assignments

1. Without resorting to modern software packages, write a program that implements convolution—the way as set of kernels is moved along the original pixel matrix, which expressed the original data using higher-level features. The motivation is to force yourself into thinking through all necessary details of convolution's working.
2. Without resorting to modern software packages, write a program that implements pooling (both max-pooling and ave-pooling). Add this program to the one from the previous question, and do in a way that is flexible enough to enable experimentation with the most diverse architectures.
3. Modern programming language make implementation of deep learning much easier than older tools. Among the currently most popular, Python definitely has to be mentioned, alongside with its superstructures such as PyTorch or TensorFlow. Go to your browser and find more information.
4. If you implement a working version of CNN, apply it to the MNIST data available from the web.[4]

[4] http://yann.lecun.com/exdb/mnist/.

Chapter 17
Reinforcement Learning: N-Armed Bandits and Episodes

The field of *reinforcement learning* studies techniques that allow a computer program to respond to signals about an environment's states and to react adequately. This is very practical. In this way, the computer can learn how to navigate a complicated maze, how to balance a broom-stick, and even how to drive a car or how to play complicated games such as chess or Go. The principle is to "learn from experience." Facing diverse situations, the agent experiments, acts, and receives for its actions rewards or punishments. Based on these, it optimizes its behavior so as to maximize the anticipated rewards.

Scientists have developed a broad family of useful algorithms, from simple manipulation of lookup tables all the way up to *Deep-Q-learning* and other advanced approaches. So rich is the material that two full chapters are needed to cover it. The current one introduces the problem and acquaints the reader with the entry-level *episodic* formulation, explaining how to apply it to simple tasks. More advanced techniques are relegated to the next chapter.

17.1 Addressing the N-Armed Bandit Problem

Let us begin with a highly simplified version of the task. We will get used to the terminology and to the principles forming this paradigm's foundations.

N-Armed Bandit Figure 17.1 shows five slot machines. Whenever a coin is inserted into any of them, and the lever is pulled down, a certain sum, different at each trial (and very often just zero), is returned. Suppose that each machine is known to be different; some give higher returns, others smaller, but nobody knows what their individual averages are. It is up to the gambler to find out.

The slot machine's tendency to rob you of your hard-earned money is the reason behind its unflattering nick-name, *one-armed bandit*. In the case just described, there are N of them, and this is why we will refer to them as the *N-armed*

Fig. 17.1 N-armed bandit: which machine offers the highest average return?

bandit. Introductory texts on reinforcement learning are fond of this domain whose conceptual simplicity makes the basic issues easy to explain.

Terminology Instead of "gambler," we will use the neutral term, *agent*; and the choice of a concrete slot machine is an *action*. In Fig. 17.1, the number of actions equals the number of machines. If we label the machines with integers, $1, 2, 3, 4$, and 5, then the symbol a_i will indicate the i-th action, the one that places the bet on the i-th machine.

The same action may receive a different reward each time it is selected. The *best action* is the one that promises the highest *average return* in the long run. Generally speaking, some returns can be negative, in which case they are sometimes called "punishments"; the sign, however, does not affect the essence of the machine-learning techniques.

***Greedy* Strategy** A statistician's first instinct is simple: let the agent try each action a great many times, keep a tally of the individual returns, and then calculate the averages. Once these are known, the agent should forever stay with the machine with the highest average return.

On the face of it, this appears to make sense. However, we must not forget that each action takes time and costs a quarter. In a domain with one thousand slot machines instead of just five, systematic experimentation is bound to be so costly and time-consuming as to be not only impractical, but downright silly.

Flexible ϵ-*Greedy* Strategy A more pragmatic approach will start with limited experience, perhaps based on just a few tests for each action. The agent will stick with the action that these tests have shown to be best—but not always! Admitting that the choice rests on imperfect knowledge, the agent will occasionally experiment with the other actions, too. This experimentation may show some other machine to be best.

Put more technically, the agent will combine *exploitation* of the apparently best action with *exploration* of the alternatives. Exploitation dominates, exploration is rare, its frequency being controlled by a user-specified parameter, ϵ. For instance, $\epsilon = 0.1$ means that the best action is chosen 90% of the time, and in the remaining 10% cases, a randomly selected non-best action is preferred.

Note the difference. Whereas the *greedy* strategy always chooses the best action, the ϵ-*greedy* strategy interleaves exploitation with exploration.

Table 17.1 Choosing the action by the ϵ-greedy strategy

Input: User-specified parameter, ϵ (usually a small number such as 0.05)

1. Generate a random number, $p \in (0, 1)$, from the uniform distribution.
2. If $p \geq \epsilon$, take the best action (*exploitation*).
3. If $p < \epsilon$, take an action chosen randomly from the remaining, non-best actions (*exploration*).

Implementing the ϵ-Greedy Strategy The principle is summarized in Table 17.1. The user provides the value of $\epsilon \in (0, 1)$. Each time an action is to be chosen, a random number, $x \in [0, 1]$, is generated from the uniform distribution. If $x > \epsilon$, the best action is taken (exploitation). If $x \leq \epsilon$, some other action is preferred (exploration). In exploration, any "non-best" action has the same chance of being selected.

Role of ϵ The concrete behavior of the ϵ-greedy strategy depends on the choice of ϵ. A relatively high value, say, $\epsilon = 0.1$, implies frequent exploration (10%, in this specific case). In the case of a mistaken assumption about which action is best, this frequent exploration will soon bring about an improvement. This is good. On the other hand, if the current opinion *is* correct, then the consequence of $\epsilon = 0.1$ is that the agent is forced to pick a sub-optimal machine in 10% of all trials—and this is *not* what we want.

In practical implementations, it often makes sense to start with relatively frequent exploration, so long as the "currently best" action is rather arbitrary. Later, when the experimentation has provided convincing evidence, ϵ can be reduced.

Non-stationary Domains In some applications, the system's properties can vary in time.[1] For a certain period, action a_i is best, but then the situation changes in the sense that some $a_j (i \neq j)$ starts giving higher returns than a_i. In non-stationary domains of this kind, higher values of ϵ have to be considered to allow the agent quickly to adapt to the changed circumstances.

In some domain, it is quite obvious whether or not the system is stationary. In others, it is up to the machine-learning program to figure it out automatically.

Keeping a Tally of the Rewards In the course of the experiments, the agent keeps a tally of the received returns for each of the available actions (do not forget that these returns are stochastic: the same action can incur a different reward each time the action is taken). From these returns, averages are calculated. Suppose that action a_i has been taken three times, with the following returns: $r_1 = 0, r_2 = 9$, and $r_3 = 3$. The average return is then $Q(a_i) = (r_1 + r_2 + r_3)/3 = (0 + 9 + 3)/3 = 4$. Here, the symbol $Q(a_i)$ stands for the *q*uality of a_i.

[1]The reader will recall that this possibility was discussed in the context of classifier-learning techniques in Sect. 11.4.

The simplest implementation will rely on a lookup table whose i-th row contains the list of the rewards returned so far by the i-th action. Each time the action is taken, the reward is entered in the corresponding row of the table, and the average is recalculated. This, of course, is not very efficient. The next section will present a better way of doing it.

17.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Describe the N-armed bandit problem. Explain what is meant by an *action* and its *reward*. What is the agent's goal?
- Describe the *greedy* strategy and the ϵ-*greedy* strategy. What is *exploitation* and what is *exploration*? How would you implement these strategies?

17.2 Additional Information

To set the stage for the rest of this chapter, the previous section constrained itself to basic principles. For practical purposes, though, certain improvements are necessary.

No Need to Store All Older Returns The programmer can take advantage of the following formula where $Q_k(a)$ is action a's quality, defined as the average return over k experiments, and r_{k+1} is the reward at the $(k+1)$st decision to take this action.

$$Q_{k+1}(a) = Q_k(a) + \frac{1}{k+1}[r_{k+1} - Q_k(a)] \tag{17.1}$$

The reader can easily verify the correctness of this last equation by substituting to it the following two definitions:

$$Q_k(a) = \frac{1}{k}\sum_{i=1}^{k} r_i \qquad\qquad Q_{k+1}(a) = \frac{1}{k+1}\sum_{i=1}^{k+1} r_i$$

Equation (17.1) makes it possible to update the information about the action's quality based only on the newly received reward, r_{k+1}, and the value of k. This means that it is enough to store only k and $Q_k(a)$; the older rewards are not needed.

Lookup Table One way to implement the lookup table is to provide one row for each possible action. Each row contains the information about the number of times,

Table 17.2 The lookup table for a simple implementation of reinforcement learning

Suppose that only five different actions are possible, a, b, c, d, and e. The lookup table contains for each action the number of times it has been taken, and the current estimate of the average return.

Action	# trials	Quality
a	$k_a = \cdots$	$Q_{k_a}(a)$
b	$k_b = \cdots$	$Q_{k_b}(b)$
c	$k_c = \cdots$	$Q_{k_c}(c)$
d	$k_d = \cdots$	$Q_{k_d}(d)$
e	$k_e = \cdots$	$Q_{k_e}(e)$

Each time an action is taken, the k and Q in the corresponding row are updated; k is incremented by one, and Q is recalculated using Eq. (17.1).

k, the corresponding action has been taken, and the current value of the average reward, $Q_k(a)$. This is illustrated by Table 17.2 for the simple case of the five slot machines from the previous section. One may add also the information about the action that currently appears to be best.

General Format of the Learning Formula Let us simplify Eq. (17.1) by replacing the coefficient $1/(k+1)$ with α_{k+1}:

$$Q_{k+1}(a) = Q_k(a) + \alpha_{k+1}[r_{k+1} - Q_k(a)] \tag{17.2}$$

With a grain of salt, we can say that the entire theory of reinforcement learning is nothing but a study of diverse variations on this theme. It is thus important to understand how it works.

The reader already knows that action a's quality, $Q_k(a)$, is the *estimate* of its average reward, based on the evidence provided by the first k instances when a was taken. If the reward at the $(k+1)$th step is higher than the current estimate, $r_{k+1} > Q_k(a)$, then Eq. (17.2) will increase the estimate. In the opposite case, $r_{k+1} < Q_k(a)$, the value of the estimate is reduced.

Diminishing Returns: Good or Bad? With the growing number of times an action has been taken, the value of the coefficient $\alpha = 1/(k+1)$ becomes smaller. For instance, if $k = 9$, then $\alpha = 1/(9+1) = 0.1$, and Eq. (17.2) becomes $Q_{k+1}(a) = Q_k(a) + 0.1[r_{k+1} - Q_k(a)]$. But when $k = 99$, we will have $\alpha = 0.01$, which means the term that modifies the estimate will be so small as to make the change in $Q_{k+1}(a)$ almost imperceptible. Put another way, α keeps giving smaller and smaller weights to later rewards.

On the face of it, these *diminishing returns* make sense: the more experience has gone into the average return's estimate, the higher our confidence in its correctness, and the lower our willingness to modify it after some scant additional evidence.

The situation is different in *non-stationary domains* where an action's desirability is subject to changes in time. Small values of α then make the agent excessively conservative, preventing any reaction to the changed situation. This is why engineers often do not want α to become too small; instead, they prefer to set it to some constant value, say, $\alpha = 0.2$.

Optimistic Initialization Equation (17.1) updates $Q_{k+1}(a)$ based on its previous value, $Q_k(a)$. To be able to begin somewhere, we need for each action its initial value, $Q_0(a_i)$, from which the learning process will start. This initial value can be suggested by early experimentation that has tried each action a limited number of times.

However, a more elegant solution exists, known as *optimistic initialization*: before any learning begins, we set all estimates to a value known to exceed any single return (this is why it is called "optimistic"). For instance, if all returns are expected to fall between 0 and 10, then this approach may begin with all actions having $Q_0(a_i) = 50$.

At an exploitation step, the learner chooses the action with the highest Q, breaking ties randomly. Since all actions have the same initial value, they have the same chance of being taken. Suppose that a_i is picked. Since the received reward is smaller than the initial value, Eq. (17.1) will reduce this action's value. In the next exploitation step, some other action will be selected, and its value reduced, too, and the process will continue until all actions have been tried.

Numeric Example Table 17.3 illustrates this technique's the behavior. In this particular example, the lookup table represents each action by one column. At the beginning, all actions have $k = 0$ because none of them has been tried, yet; the estimates of average rewards are all set to $Q_0(a_i) = 50$ which is believed to be more than any possible reward that any action may draw.

The first step chooses at random action a_3 and receives reward $r = 4$. Equation (17.1) reduces the value estimate to $Q_1(a_3) = 50 + \frac{1}{0+1}(4 - 50) = 4$ which is less than the value of any other action. As a result, the next step will use a different action. Suppose it is a_5 and suppose the reward is $r = 2$. Equation (17.1) reduces the value estimate to $Q_1(a_3) = 50 + \frac{1}{0+1}(2 - 50) = 2$. This means that two actions have already been tried. The reader can see that *optimistic initialization* forces the system to try, at the beginning, all actions more or less systematically, as long as *exploitation* is being used.

If, at some later stage, action a_3 is taken again, with reward, say, $r = 6$, the quality is updated with $k = 1$ so that $Q_2(a_3) = 4 + \frac{1}{1+1}(6 - 4) = 5$. We can see that now the change is smaller than at the previous step.

Soft-Max ϵ-greedy strategy has an alternative known as *soft-max*. Chapter 6 presented a mechanism capable of making neural-network's outputs sum to 1 so that they can be interpreted as probabilities. Here is how the same "trick" is employed in the context of reinforcement learning.

Table 17.3 Optimistic initialization of the rewards in the N-armed bandit game

Consider a system permitting five different actions. Suppose that the maximum possible reward for each of them is $r = 10$. Initialize the estimates of all actions to $Q_0(a_i) = 50$, a value that exceeds the maximum possible reward:

	a_1	a_2	a_3	a_4	a_5
$Q_0(a_i)$	50,k=0	50,k=0	50,k=0	50,k=0	50,k=0

All actions having the same value, the first action is chosen randomly. Suppose that a_3 is taken, and that the reward is $r = 4$. In the third column, k is incremented and the quality updated using Eq. (17.1): $Q_1(a_3) = 50 + \frac{1}{0+1}(4 - 50) = 4$. The other columns are left unchanged.

	a_1	a_2	a_3	a_4	a_5
$Q_0(a_i)$	50,k=0	50,k=0	50,k=0	50,k=0	50,k=0
$Q_1(a_i)$	50,k=0	50,k=0	4,k=1	50,k=0	50,k=0

For the next action, a_3 cannot be selected because all other actions have higher value, 50. Suppose that a_5 is taken, with reward $r = 2$. In the fifth column, k is incremented and the quality updated using Eq. (17.1): $Q_1(a_5) = 50 + \frac{1}{0+1}(2 - 50) = 2$.

Here is the table's new version:

	a_1	a_2	a_3	a_4	a_5
$Q_0(a_i)$	50,k=0	50,k=0	50,k=0	50,k=0	50,k=0
$Q_1(a_i)$	50,k=0	50,k=0	4,k=1	50,k=0	50,k=0
$Q_2(a_i)$	50,k=0	50,k=0	4,k=1	50,k=0	2,k=1

Again, the next action is selected from those that have not yet been tried and still have the initial high values.

The reader can see that the *optimistic initialization* indeed leads to systematic exploration of all actions at the beginning.

Suppose that, for action a_i, the lookup table gives the average-reward estimate $Q(a_i)$, and suppose that z is a parameter whose value has been set by the user. The probability with which action a_i will be selected is calculated as follows:

$$P(a_i) = \frac{z^{Q(a_i)}}{\sum_j z^{Q(a_j)}} \tag{17.3}$$

It would be easy to show that, as in Chap. 6, the sum of these probabilities over all actions is 1. Note that reinforcement learning is used to somewhat higher flexibility, in the formula. Whereas neural networks typically assumed $z = e = 2.72$ (a number known from natural logarithms), here it is the user who decides what the base of the exponential function should be.

Numeric Example Suppose there are three different actions to choose from, and let their Q-values be $Q(a_1) = 5$, $Q(a_1) = 8$, and $Q(a_1) = 3$. The following table summarizes the individual actions' probabilities calculated by Eq. (17.3) for different values of z.

	a_1	a_2	a_3
$z = 2$	0.11	0.86	0.03
$z = 5$	0.0086	0.9911	0.0003
$z = 0.5$	0.194	0.025	0.781

Note that higher values of z tend to concentrate all the probability at the best action. The probabilities in the case of $z = e$ would be somewhere between those in the rows for $z = 2$ and $z = 5$. Note also that for $z < 1$, the behavior is exactly opposite from what we need: the best action is given the smallest probability.

It is thus always necessary to choose $z > 1$!

Selecting the Next Action with *Soft-Max* To select the action, the values in the first row of the previous table are used in the following manner. Divide the interval [0, 1] into three sub-intervals whose lengths are 0.11, 0.86, and 0.03, respectively. Each sub-interval represents one of the three actions. Generate a random number (uniform distribution) from the interval [0, 1]. Choose the action into whose sub-interval the number falls.

In the case of n actions, n sub-intervals are used, but the principle is the same.

Reinforcement Learning Algorithm Table 17.4 shows how to apply *reinforcement learning* to the N-armed bandit. After *optimistic initialization*, the actions are selected by a user-specified strategy, ϵ-greedy of *softmax*. Whichever strategy is used, the user has to specify the parameters: ϵ or the z (the latter in the case of *softmax*). Both parameters can vary in time. Each action results in a reward which is then used to update the selected action's Q-value (average-reward estimate).

The algorithm is written as an endless loop. In reality, it is usually run a predefined number of times.

Table 17.4 *Reinforcement learning* and the N-armed bandit

```
Input: a set of actions, $a_i$;
       user-specified strategy ($\epsilon$-greedy or softmax) to choose the next action.

1. Carry out optimistic initialization and set $k_i = 0$ for $\forall i$.
2. Choose an action and denote it by $a_i$.
3. Carry out $a_i$ and receive reward $r_i$.
4. Recalculate the average-reward estimate for this action:

       $Q_{k_i+1}(a_i) = Q_{k_i}(a_i) + \frac{1}{k_i+1}[r_i - Q_{k_i}(a_i)]$

5. Set $k_i = k_i + 1$ and return to step 2.
```

17.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What information is stored in the lookup table? Write down the formula for updating the Q-value. What else has to be updated after an action has been carried out?
- Explain the motivation behind *optimistic initialization*. How does it work?
- What is meant by *diminishing returns*? Under what circumstances are they beneficial, and under what circumstances do we prefer the returns not to diminish?
- Explain how *soft-max* is used to select the next action. What is the impact of its parameter z? What values for z would you recommend and why?

17.3 Learning to Navigate a Maze

The N-armed bandit problem has helped us explain some elementary principles underlying *reinforcement learning*. Now that the reader has developed their basic understanding, we can move on to a more realistic setting.

Maze Figure 17.2 shows a very simple version of a popular puzzle: a maze. The task is to enter the maze at location denoted by S, and, moving one square at a time, to find the shortest path to G. Obviously, the trick is to discover that, after the first few steps to the right, one has to turn down and not up.

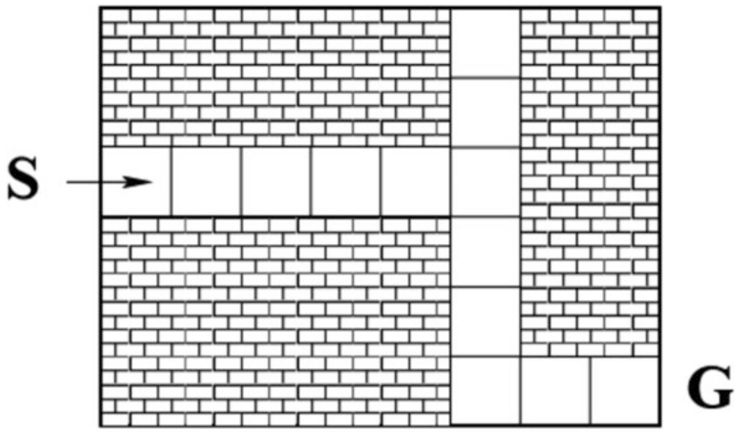


Fig. 17.2 The agent starts at S; it wants to find the shortest path to G

States and Actions Let us now formulate the problem in the terminology of *reinforcement learning*. Each square in the maze is a *state*. At each state, the agent has to choose a concrete *action*. In principle, four actions are possible: *up, down, left*, and *right*. In some states, however, some of these actions are illegal (those that would "hit the wall"). The result of an action is that the agent has moved to another square, which means that its state has changed.

Importantly, each state represents one N-armed bandit from the previous sections in the sense that we want the agent to identify for each state the *best action*: the one to be taken if the agent is to follow the shortest path.

Lookup Table for the Maze As before, the agent will rely on a lookup table which, however, is bigger than the one employed in the plain N-armed bandit. This is because we need to list all states; and for each state, all actions legal in this state. Besides, the table has to store the information about which state is reached as a result of a given action. For instance, after action *right* in the very first square in Fig. 17.2, the agent reaches the square to the right of the current one.

An example of such a table, still somewhat simplified, is shown in Table 17.5. The number of rows is much higher than in the basic N-armed problem but, if the maze is simple, this is manageable. Chapter 18, however, will introduce domains where the table is so large (billions of rows or more) that it cannot reasonably be used.

Rewards In the maze problem, filling out the values in Table 17.5 is quite straightforward—with one exception. To update an action's quality, we need the reward. In N-armed bandit, the reward followed the action immediately. In the case

Table 17.5 An example lookup table for the maze

For each state, all actions are listed. For each state and each action, the table gives the current estimate of the average reward (*quality*, Q), the number of times, k the action has been taken in this state, and the new state that results from the application of this action. For each state, all actions are listed. For each state and each action, the table gives the current estimate of the average reward (*quality*, Q), the number of times, k the action has been taken in this state, and the new state that results from the application of this action.

State	Action	Quality	k	New state
s_1	$a_{1,1}$	$Q(a_{1,1})$	3	s_2
	$a_{1,2}$	$Q(a_{1,2})$	4	s_3
s_2	$a_{2,1}$	$Q(a_{2,1})$	1	s_3
	$a_{2,2}$	$Q(a_{2,2})$	1	s_4
	$a_{2,3}$	$Q(a_{2,3})$	3	s_5
⋮	⋮	⋮	⋮	⋮
s_N	$a_{N,1}$	$Q(a_{N,1})$	4	s_9
	$a_{N,2}$	$Q(a_{N,2})$	1	s_4
	$a_{N,3}$	$Q(a_{N,3})$	2	s_7

of the maze, this is not so simple. Most of the time, to establish an action's reward before the goal has been reached is impossible, or at least speculative.

Episode In the earliest stages, before any learning has begun, the agent's walk through the maze is blind and random, especially if *optimistic initialization* assigned to all actions the same initial values. But even random walk will sooner or later reach the end: an agent that has started at square S will land in the goal denoted in Fig. 17.2 as G. Each such journey from S to G will be called an *episode*.

Episode's Reward In the course of the episode, some actions are lucky in the sense that, by mere chance, a good choice was made—one that reaches the goal quickly. Thus after the first step, the agent may choose to move to the right instead to the left. This is good because moving to the left would only bring the agent back to the start, S, which would hardly be regarded as an efficient path. If most actions are lucky, the goal is reached faster than in the case of less fortunate choices. The reader will agree that the quality of the selected actions is reflected in the number of steps that had to be taken before arriving at G.

If the agent has completed the walk from S to G in N steps, the *overall reward* for the given episode can therefore be established as follows:

$$R = -N \tag{17.4}$$

The greater the number of steps, the "more negative" the reward (the greater the punishment). This means that the reward is maximized when the number of steps is minimized.

Episodic Formulation of *Reinforcement Learning* A simple way to implement *reinforcement learning* in the maze problem maintains the list of all actions, $L_{actions}$, that were taken in the course of the episode. Once the agent has reached G, one can calculate the episode's reward, R, which is then used to update the Q-values of every single action listed in $L_{actions}$.

The algorithm to carry out one episode is summarized in Table 17.6. Repeating the procedure for many episodes gradually improves the Q-values. This leads to better choices of the actions, and to shorter walks of the agent. In other words, the technique thus described is likely to find a short path through the maze.

Numeric Example Let us illustrate the technique's behavior on the same maze as before. In Fig. 17.3, the state at the "intersection" is marked by a star. When at this state, the agent can choose from three actions: *left, up*, and *down*. Suppose that all of them were initialized equally, with $Q_0(a_i) = 0$.[2]

All actions having the same Q-value, the first time the agent finds itself in this state, the choice is random. Suppose the agent selects *up*. This being an unfortunate

[2] Since all rewards are known to be negative, $Q_0(a_i)$ is clearly greater than any episode's reward.

Table 17.6 One episode in the maze

```
Input: The maze and its lookup table;
       The strategy ($\epsilon$-greedy or softmax) to choose the next action.
       An empty list of actions, $L_{actions}$.

1. Let $t = 0$. Let $S$ be the first state, $s_0$.
2. When in state $s_t$, choose an action, $a_t$, and add it to $L_{actions}$.
3. Apply $a_t$, reaching state $s_{t+1}$ that is given in the lookup table.
4. If $s_{t+1} \neq G$, set $t = t + 1$ and return to step 2.
5. If $s_{t+1} = G$, calculate the episode's reward, $R$, and use it to update the $Q$-values of all actions
   listed in $L_{actions}$.
```

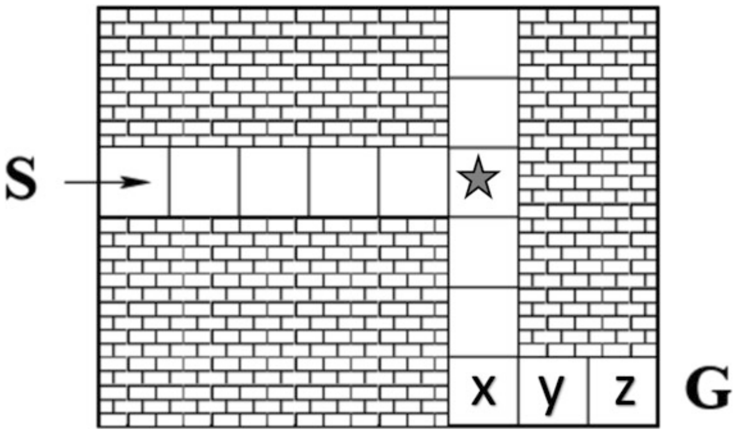


Fig. 17.3 At the state marked by a star, the agent can choose from three different actions: *left, up*, and *down*

turn to take, the entire walk will take quite a few steps, say, 40, which means $R = -40$. Using this in Eq. (17.2), we get the following Q-update:

$$Q_{up} = 0 + \frac{1}{0+1}(-40 - 0) = -40$$

In the next episode, suppose that the agent, when finding itself in the same location, chooses *down*, and suppose that the entire walk then takes 20 steps, which means $R = -20$. Updating Q_{down} with Eq. (17.2) then results in the following Q-update:

$$Q_{down} = 0 + \frac{1}{0+1}(-20 - 0) = -20$$

In these two episodes, the one with *down* was shorter than the one with *up*, and this resulted in $Q_{down} > Q_{up}$. In the future, ϵ-greedy strategy of *soft-max* is likely to go *down* instead of *up*.

17.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What information has to be stored in the lookup table if we want to solve the *maze*?
- Explain what is an *episode* in the *maze* application. Give an example of how to establish the episode's reward.
- How do we calculate the rewards of the individual actions in the *episodic formulation* of *reinforcement learning*? Write down the formula.

17.4 Variations on the Episodic Theme

Now that the reader understands what the episodic formulation of *reinforcement learning* is all about, we can take a closer look at it.

Some Criticism The main advantage of the episodic formulation is its simplicity. On the other hand, we must not overlook a serious shortcoming: all actions in $L_{actions}$ receive the same reward, which clearly is not right because not all of them have contributed equally. Some were ideal, for the given state, some less so, and some were perhaps outright bad—and yet they all are rewarded with the same R!

In the course of a great many episodes, these "injustices" get averaged out. The reason why some episodes are very long is that most of the actions are bad and are thus punished severely. This is unfair to the good actions, but these are compensated by the statistical expectation they will occur in much shorter walks in which the bad actions do not participate.

Still, one cannot help suspecting that this way of handling the maze problem is less than optimal. Chapter 18 will investigate alternative solutions that tend to outperform the episodic approach significantly.

Immediate Rewards? The criticism just mentioned would disappear if we knew how to determine the reward immediately after each action. This is indeed sometimes possible; and even in domain where this is *not* possible, one can often at least make an estimate. In that event, the learning process will be improved in the sense that each action's Q-value be updated "according to the action's own deserts," instead of all actions receiving the same episodic reward. This, too, will get more attention in Chap. 18.

Discounted Returns The reward-calculating Formula (17.4) followed a simple principle: the longer it takes to complete the walk through the maze, the smaller the reward should be (higher value of N lead to a smaller $R = -N$). The same effect can be achieved by the so-called *discounted returns*.

In this case, the ultimate reward for completing the walk is $R = 1$. The reward assigned to any concrete action from $L_{actions}$ then depends on the number of steps between this action and the goal, G. If there are k steps remaining, the value of the given action is updated using the following formula were $\gamma \in (0, 1)$ is a user-set discounting constant:

$$r_k = \gamma^k R \tag{17.5}$$

Note how the growing value of k decreases the coefficient of R. If the goal is reached in 10 steps, then the discounted reward for $\gamma = 0.9$ is $R = 0.9^{10} \cdot 1 = 0.35$. The engineer has to be aware of the possibility of "underflow" of r_k if k is extremely high. On the other hand, r_k is always positive, whereas the formula from the previous section always leads to negative rewards.

Concrete Implementation of Discounted Returns Previous paragraph suggested that the k in the *discounted return* formula should be the number of steps taken between S and G. The same end-of-episode return is applied ($R = 1$), but when the Q-values of the actions from $L_{actions}$ are modified, the reward is discounted by Eq. (17.5) (for Q-updates, r_k is used instead of R).

A more flexible alternative will use for k an even more appropriate value: the number of steps between the current state and G. This information is easily obtained from the contents if $L_{actions}$: it is simply the number of entries between the current state and the end of the list.

There is one important detail to consider. In the course of the learning process, the same location in the maze can be visited more than once, and the same action may be taken. If this is the case, the user has to decide whether to use for k the number of steps (from the given state to G) after the *first visit* to this state or from the *last visit* to this state.

Numeric Example Table 17.7 shows how the rewards may be assigned in the case of the *first-visit* variation. The table contains only an excerpt from the history of a single episode. We assume that before these entries in the table, none of the three mentioned state–actions, x, y, and z, has been taken (Fig. 17.3 shows which actions these are).

The second row shows the values of the discounted reward for the state–actions. Thus the first occurrence of x in the table receives $r = 0.48$ because it took the agent another $k - 7$ steps before it reached G. Using $\gamma = 0.9$, the reward for x is therefore calculated as $r = 0.9^7 = 0.48$. In the course of the random walk, x is later revisited, but this has no impact on the action's reward if it follows the *first-visit* principle.

If the *last-visit* principle is used, we realize that to reach G from the last occurrence of x, only five steps are needed. This means that the reward for x would then be $r = \gamma^5 = 0.59$.

Table 17.7 End of one episode in the maze. The discounted rewards are assigned on the *first-visit* principle

The first row of the table shows the actions taken by the agent at the last few steps of the random walk, G being the goal. Note that some of the state–actions are visited more than once. However, let us assume that none of the three (x, y, and z) has been visited before the beginning of this section of the table.

state	...	x	y	x	y	z	y	z	G
r_k	...	0.48	0.53	–	–	0.73	–	–	–

The second row shows the values of the discounted rewards for the actions, using $\gamma = 0.9$, and assuming the episode's overall reward being $R = 1$.

For instance, from the first occurrence of x at the beginning of the table $k = 7$ more steps were taken before G was reached. Therefore, the reward is $r_k = \gamma^k R = 0.9^7 = 0.48$ (after rounding).

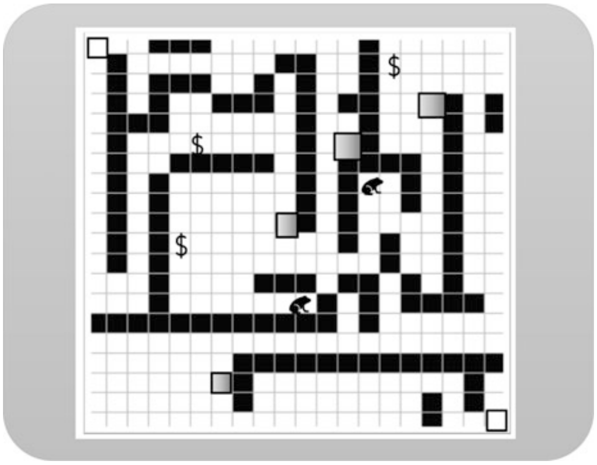


Fig. 17.4 More advanced versions of the "maze" problem may include potholes, monsters, and caches. These may affect the reward

More Sophisticated Maze Figure 17.4 shows a more challenging maze than that of the one we have been working with so far. For one thing, it is larger, and thus more difficult to solve. For another, certain extra functions have been added. Some squares represent "potholes" that slow down the agent's walk (increasing the overall time needed to reach G). Others contain "caches" where the agent can collect monetary rewards. Yet others are occupied by "monsters" that either have to be bribed or perhaps even can destroy the agent. The point is, the rules of the game can be much more complicated than in the toy domain above.

Importantly, all of these added functions may affect the way the final reward for the episode is calculated. For instance, this reward may depend not only on the number of steps before G has been reached but also on the amount of money gained or lost during the episode. The programmer will then have to design a special formula to calculate R. The rules of the game may have to be modified, too. For instance, hitting the square occupied by a monster may cause a "stop with failure"; the agent never reaches G and may be penalized with a large negative reward.

Perhaps even the ultimate goal may be modified. Instead of finding a short path, one may require the agent to collect maximum amount of cash, but only if the entire walk does not exceed 100 steps.

17.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Discuss the advantages and shortcomings of the *episodic* formulation of *reinforcement learning*.
- What is the motivation behind the idea of *discounted returns*? Give the precise formula, and discuss the impact of γ and k.
- Discuss the alternative possibilities of adjusting k for *discounting rewards* in the *maze* problem.

17.5 Car Races and Beyond

It is one thing is to grasp the basic principles; it is another to apply them creatively to a realistic problem. To gain more experience, let us take a look at some other tasks that can benefit from *reinforcement learning*.

Car Races: Problem Definition Figure 17.5 shows a small segment of a track for car races. The agent is driving. At each moment, its car finds itself on one of the squares of the track and is moving with a specific speed in a specific direction. The agent can turn the steering wheel, accelerate, and decelerate. Of course, how exactly

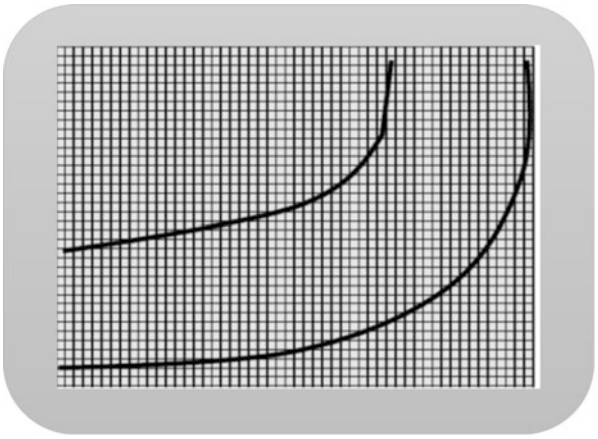


Fig. 17.5 In the "racing car" problem, the agent wants to find the ideal speed and direction when making the curve

it does cannot be arbitrary. If the car goes too fast, it is unable to make the turn and is bound to crash in the wall. If it goes too slowly, it will lose the race.

The goal for *reinforcement learning* is to find an ideal way of operating the car so that it makes the track as fast as possible without crashing.

States and Actions A state is here defined by the car's location and velocity. The location is defined by a pair of integers, $[x, y]$, where x is the horizontal coordinate and y is the vertical coordinate. Velocity is a pair of integers, $[v_x, v_y]$, again one for each coordinate.

One step in the agent's driving is defined as a change in the location according to the following formulas where $t = 1$ represents one time unit:

$$x = x + v_x t = x + v_x$$

$$y = y + v_y t = y + v_y$$

An action means to change velocity. For instance, the agent can change v_x or v_y or both by 1 or -1 or 0 under the constraint that the velocity along either coordinate is not allowed to be greater than 6 or smaller than -6.

There are a great many ways of designing the lookup table. What they have in all in common is that the table will be very large if it is meant to contain all states, and all actions that are possible in these states. A more economic alternative—updating only states, not actions—is discussed in the next section.

Rewards As in the *maze* domain, the learning program in *car races* can simply count the number of steps that the agent has needed to reach the end of the track (or track segment) from the location of the start. If the car hits the wall, a very high punishment (negative reward) can be issued.

Many other mechanisms can be used, though. The agent may get extra punishment for getting too close to the wall. Another possibility is to abandon the principle of plain counting of the number of steps (recall the formulas, $R = -N$ and $r = \gamma^N R$) and instead assign different weights to steps in which the velocity is increased than to steps in which the velocity is decreased—this will encourage "aggressive driving." The engineer has to be creative; an ingenious way of defining the rewards can strongly impact the efficiency of the learning process, and the agent's performance in general.

Look-Ahead Strategy The race-track segment from Fig. 17.5 is a simplification meant for instructional purposes. A more realistic problem statement will want the car to optimize its driving along the entire closed loop track. This, of course, is much more challenging computationally because of the great number of states and actions.

The computational costs can be reduced by a look-ahead strategy. Rather than storing all states and actions in a huge lookup table, the agent focuses only on a segment such as the one in Fig. 17.5. Before starting, it first attempts "in its head" N tentative episodes (with N being an adjustable parameter, a reasonable value being a few hundred), gradually learning from them. Then it completes this segment, puts this lookup table aside, and starts a new one for the next segment, and on, until the entire track is finished.

Alternative Goals What do we actually want to achieve? Of course, the primary task is to learn how to complete the track without crashing. However, some other goals can be considered. We may want the agent to complete the track in a minimum number of steps (as fast as possible), we may want to find the shortest path, and so on. The experimenter may want to observe how these goals are affected by the number of tentative episodes (N), by the length of the "look-ahead," and by the creative use of rewards.

Stochastic Nature of *Reinforcement Learning* For the sake of simplicity, the formulations of the two test-beds, *maze* and *car-race*, were intentionally simplified: calculation of the rewards was straightforward and deterministic; and each action resulted in a concrete next state. Such simplification was necessary for an easy-to-understand presentation of the basic principles.

In reality, many applications are much more challenging. In many domains, the result of any action (the next state) cannot be determined in advance. For instance, in game playing, the next state that the agent is going to face after it has made a move depends also on the response chosen by the opponent.

The rewards themselves are often stochastic in a way reminiscent of the N-armed bandit from the beginning of this chapter.

Stochastic Behaviors in Classroom Projects In the maze problem, the rewards of certain situations can be generated by a random-number generator. This can be the case of delays caused by hitting potholes, monetary rewards provided by the discovered caches, or the consequences of encountering monsters.

In the *car-races* domain, the next state (location) can be affected by "wind" or "spilled oil." For instance, a practical implementation may request that, with 20% probability, the agent's next location is shifted one square to the left.

What Constitutes a Good Application for *Reinforcement Learning*? The reader has by now developed an initial idea of the characteristics that mark a domain well-suited for this machine-learning paradigm. Of course, the first requirement is that it should be possible to specify the requisite set of states and actions, and the way of establishing the rewards. Further on, *reinforcement learning* is often resorted to in domains where analytic solution is unknown; or, if it *is* known, it is so complicated or computational expensive that it is better to resort to experimentation. Analysis can also be impossible if the system we want to control suffers from a great degree of randomness.

This said, one must not forget that any engineering problem can be addressed in alternative ways; the engineer thus needs to be sure that *reinforcement learning* is indeed more appropriate for the given task than competing approaches.

All these characteristics are typical of many popular games such as Go, Backgammon, or chess. For these, however, the episodic approach in the simple version presented by this chapter is somewhat inflexible; more sophisticated techniques, such as those from Chap. 18), are called for.

Where do the Rewards Come from? In the toy domains typically addressed by classroom projects, the rewards are easy to determine by a predefined formula or algorithm. In a real-world application, the situation is rarely so straightforward.

True, in many domains, the reward or punishment is provided by the concrete environment. Thus in the case of a chemical reactor, it may be easy to measure the time a certain chemical reaction has taken, or the weight of the final product that has thus been created. On the other hand, if a certain action leads to an explosion, using this action for machine-learning purposes would hardly be desirable. Likewise, training a self-driving car will rarely exploit many crashes.

In situations of this kind, the training will preferably rely on a computer model that can crash any number of times without sinister consequences. However, such models are often only crude approximations of the underlying systems. Very often, therefore, the model is used only to obtain the first approximation to be later fine-tuned in the real world.

17.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How would you implement the *car races* problem? What will be the states, actions, and rewards? How would you make this domain stochastic?
- What constitutes an appropriate application for *reinforcement learning*?
- Where do the rewards come from in a realistic situation? Discuss the possibility of receiving the rewards from an existing system versus from the system's computer model.

17.6 Practical Ideas

Now that the reader has an idea of how to implement the baseline version of the episodic task, it is time to consider a few additional "tricks" that make the learning process more efficient.

Maintaining States, Not Actions In the *car races*, as in many other domains, keeping a tally of the Q-values of all actions in all states may prove impractical. In similar situations, engineers therefore work with lookup tables that maintain only the values of states and not actions. Each row represents one state, and contains this state's value: the estimate of the average returns achieved by the agent's passing through this state. To prevent confusion, the state values are habitually denoted by V (whereas Q is reserved for action values).

The number of states being smaller than the number of actions, the lookup table thus becomes more manageable. On the negative side, the engineer may not be prepared to tolerate that the consequences of concrete actions are lost.

Episodes and State Value Updates During the episode, a list, L_{states}, of all states that the agent has passed through in a given episode is maintained. Once the end of the episode has been reached, and the reward R received, the values of all states in L_{states} are updated using the following formula.

$$V(s_{t+1}) = V(s_t) + \alpha[R - V(s_t)] \tag{17.6}$$

Note that this is essentially the same formula as Eq. (17.2). The only difference is that here we are estimating the average rewards of states, and not of actions. As before, parameter α either is $\alpha = 1/(k+1)$ or is fixed at some reasonable value, say, $\alpha \in [0.05, 0.15]$.

Simplified Lookup Table In a *reinforcement learning* system that updates state values, $V(s_i)$, instead of the state–action values, $Q(a_{i,j})$, the lookup table slightly changes from the one shown in Table 17.5. Each row, representing one state, will typically contain the value of this state, the number of times, k, the state has been visited, and the list of actions possible in this state, together with the states reached by these actions.

Choosing the Next State Suppose that at time t, the agent finds itself in a state denoted by s_t. The greedy strategy will scan the list of actions possible in s_t and learns about the states reached by these actions. Then, it would apply the action that leads to such next state that has the maximum value, max $V_{(}s_{t+1})$.

In reality, of course, we prefer to rely either on the ϵ-greedy strategy or on the *soft-max* strategy so as to give a chance to previously underestimated states.

Running One Episode Table 17.8 summarizes the learning algorithm that relies on the state-updating principles from the previous paragraphs. The agent starts at state $s_0 = S$. At each state throughout the episode, the system selects the next action using either ϵ-greedy or *soft-max* strategy, most of the time preferring the action that leads to a state will the maximum value.

Table 17.8 One episode in a program that updates state (not action) values

Input: The lookup table;
The strategy (ϵ-greedy or *softmax*) to choose the next state;
An empty list of visited states, L_{states}.

1. Let $t = 0$. Let S be the first state, s_0.
2. From the list of actions available at s_t, select one action according to the given strategy.
3. Apply the chosen action, which leads to a new state, s_{t+1}. Add s_t to L_{states}.
4. If $s_{t+1} \neq G$, set $t = t + 1$ and return to step 2.
5. If $s_{t+1} = G$, calculate the episode's reward, R, and use it to update the V-values of all states from L_{states}.

At any moment, the system maintains a list, L_{states}, of all states that have been visited during the episode. Once the final state, G, has been reached, the episode's reward is received, and this reward is used to update all states in L_{states}. These updates can be the same of all these states, but they can also be carried out on the *first-visit* or *last-visit* principles. After the updates, the list is emptied, and thus prepared for the next episode.

The reader can see that the algorithm is similar to the one employed previously, when the actions' Q-values, rather than the states' V-values, were updated.

Is the Exploration Sufficiently Exhaustive? The probabilistic natures of the ϵ-greedy or *soft-max* strategies are supposed to make each new episode different from the previous ones. Nevertheless, practical experience teaches us that in a large maze or race-track, large portions of the search space remain unexplored unless special precautions have been made. The thing is that once some "reasonably good" initial paths have been found, both strategies will tend to make the agent wander in the vicinity of this path, and rarely stray too far from it. Consequently, some states are rarely visited, and some are *never* visited.

For this reason, the learning process greatly improves if the engineer employs mechanisms capable of forcing the agent to explore the less-known regions in the search space.

Intensified Exploration: Alternative Starts One way to force the learner into more intensive exploration is to start the episode not from the "official" start, S, but from another location, preferably one that has so far been rarely visited.

The reader will recall that for each state (or for each action, if the Q-values are to be updated), the lookup table contains the number of visits (denoted in the previous sections by k). The idea of *intensified exploration* is occasionally to start the episode from one of the states with small values of k.

Flexible Uses of ϵ and z After *optimistic initialization* (Sect. 17.2), it is good to set ϵ to a high value that forces the ϵ-greedy strategy to frequent explorations. This value should then gradually be reduced once the whole search space has been sufficiently well explored. Again, whether the space has been sufficiently explored can be indicated on the values of k in the lookup table.

If the *soft-max* strategy is used, the intensity of exploration can be controlled by the value of z in Eq. (17.3). Values that only slightly exceed 1, say, $z = 1.1$, will allow frequent exploration.

Creative Use of Rewards, Actions, and States Engineers who want to employ *reinforcement learning* in realistic applications should always keep in mind that the success and failure of their efforts will often depend on how the rewards are determined.

As for the states and actions, these may appear to be straightforward—but only because the domains we have seen so far (*maze* and *car races*) were for instructional purposes made simple. In more realistic applications, the decision about what constitutes a state may be open to many alternative decisions; the concrete choice will then severely affect the system's ability to learn.

17.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the *reinforcement learning* algorithm that relies on the V-values of states, instead of on the Q-values of actions. Why is the approach sometimes preferred?
- What is meant by the observation that *reinforcement learning* in the domains similar to those from this section may fail to explore the search space adequately? Why does it happen?
- Summarize the techniques that can be used to solve the problem of inadequate exploration.

17.7 Summary and Historical Remarks

- Unlike the classifier induction systems from the previous chapters, reinforcement learning agents do not learn to classify, but to act in specific situations. They learn from direct experimentation with a system they seek to control.
- In the simple model called N-armed bandit, the agent identifies the action with the highest average return. Then it favors this best action and occasionally experiments with lower-valued action. This is carried out by ϵ-*greedy* and *softmax* strategies.
- More realistic implementations assume the existence of a set of states. Each state, modeled as an N-armed bandit, allows the agent to choose from a set of actions, each with different rewards. One of the states is denoted as the start, S, another as the goal, G.
- The problem and its *reinforcement learning* solution are illustrated by two toy domains: *maze* and *car races*. These test-beds can serve as representative models of many realistic tasks.
- The machine-learning goal is to find a series of actions that maximizes the overall reward. In the *episodic* formulation, the reward is known only when the goal, G, has been reached. This reward is then used to update the values of all states and/or actions from the episode.
- In the learning process, the system seeks to improve, for each state s and action a, the estimate of the average return observed when a is taken in state s. This estimate is denoted by $Q(s, a)$, which is why it is often called the Q-value.
- Alternatively, the system can focus on V-value. More specifically, $V(s)$ is an estimate of the average return expected when state s has been reached.
- The efficiency of the learning process can be improved by many little "tricks" and technique such as *optimistic initialization*, modified starting states, S, time-varying ϵ-values, creative rewards, etc.

Historical Remarks One of the first systematic treatments of the “bandit” problem was offered by Bellman (1956) who, in turn, was building on some still earlier work. Importantly, the same author later developed the principle of *dynamic programming* that can be seen as a direct precursor to reinforcement learning (Bellman, 1957). The basic principles of reinforcement learning probably owe most for their development to Sutton (1984). Very influential was an early summary of reinforcement learning techniques in the book by Sutton and Barto (1998).

17.8 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter’s ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

17.8.1 Exercises

1. At the end of Section 17.3, a numeric example illustrated the outcomes of two actions, *up*, and *down*, for some example rewards. Calculate the next weight update for action *left*, assuming that the walk then takes $N = 60$ steps.
2. After discounting, the reward $r_k = 1$ is reduced by the formula $R = \gamma^k r_k$ that employs the number of steps, k, taken from the action to the goal. Assuming $\gamma = 0.8$, how many steps, k, would be needed for the R to drop below 0.01?
3. Return to the example in Table 17.7. Suppose that the reward at the end of the episode is $R = 2$. What is the discounted reward of y if it is calculated on the *first-visit* principle?

17.8.2 Give It Some Thought

1. This chapter is built around the idea of using the ϵ-*greedy* and *softmax* policies. What do you think are the limitations of these policies? Can you suggest a way to overcome these limitations?
2. The principles of *reinforcement learning* have been explained using simple toy domains. Can you think of an interesting real-world application? How will you formulate it so that *reinforcement learning* can be used?
3. In the car-races domain, a lot depends on the way the engineer defines the rewards. One important aspect that did not get enough attention, in this chapter, is that these rewards should reflect the driver’s intentions. For instance, the driver prefers actions that steer the car toward the nearest curve (rather than just in

any direction). Think of ways to include these intentions in your solution. Try to suggest some other ways of making the rewards speed-up the learning process.

4. *Discounted rewards* can lead to value underflow in a realistic application in a domain where the number of steps, k, to the goal is very high. How would you prevent this from happening?
5. How would you address a domain with two or more conflicting goals? For instance, we may want to minimize the length of the maze walk, but at the same time maximize the monetary rewards provided by "caches" and stolen by "monsters." Alternatively, a rocket can minimize not only the time of a flight but also fuel consumption.

17.8.3 Computer Assignments

1. Write a program the solves the N-armed bandit. User-specified inputs should include N, the distributions of the returns provided by the individual machines (average values and standard deviations), and ϵ.
2. Design a maze that is more difficult than the one from Sect. 17.3. Write a reinforcement learning program that will address it by the episodic approach. See how the agent's "skills" gradually improve over the successive episodes. Explore how this learning ability depends on the way the rewards are assigned.
3. Design a much more complicated maze than in the previous problem. Add special features, such as monsters, potholes, caches, and so on, that cause delays, may even kill the agent, or offer gold (cashes). Write a reinforcement learning program that will address the problem by the episodic approach. The goal may not necessarily be the number of steps, but also the amount of gold accumulated during the walk.
4. Create a simple race track, perhaps just an oval. Implement a *reinforcement learning* program for the task introduced in Sect. 17.5. The goal is to find a way for the agent to complete the entire track without crashing. At a higher level, the goal can be to finish the track in minimum time. Explore the impact of different ways to calculate the reward, and the impact of diverse parameter settings.

Chapter 18
Reinforcement Learning: From TD(0) to Deep-Q-Learning

The last chapter introduces the basic principles of reinforcement learning in its episodic formulation. Episodes, however, are of limited value in many realistic domains; in others, they cannot be used at all. This is why we often prefer the much more flexible approach built around the idea of *temporal difference* and immediate rewards.

The chapter explains the principles of two basic temporal-difference techniques, SARSA and Q-learning, presenting them within the simple framework of lookup tables, paying attention to several critical details. However, in domains with many states and actions and in domains with continuous-valued variables, these simple versions are often inadequate. Engineers then rely on advanced temporal-difference techniques (such as the popular *deep-Q-learning*) that harness the power of neural networks.

Having explained all these techniques, the chapter then offers introductory information about how they help in programs that play such games as Backgammon and Go. The reader will appreciate the rich possibilities of this modern paradigm.

18.1 Immediate Rewards: Temporal Difference

An improvement over the episodic task is possible if the agent receives the reward right after each action, and not just at the episode's end. When this is possible, the more efficient *temporal-difference* learning is often preferred.

Immediate Rewards The reader will recall one serious complaint concerning the episodic task: the Q-values of all actions taken during the episode are updated with the same reward that is calculated at the episode's end. This does not seem right. Especially in the early stages of the learning process, some of these actions are beneficial whereas others are downright harmful—and yet all of them are treated in like manner. Would it not be better to give more reward to good actions and less to bad ones? The same argument applies to the scenario where the states' V-values are updated, rather than the actions' Q-values.

This indeed can be done, provided that the agent receives its reward right after each action. In that case, the action can be rewarded as it deserves, overcoming the principal weakness of the episodic task.

One has to be careful, though. In many applications, these immediate rewards are nothing but crude estimates that can do more harm than good if used dogmatically. In that event, the good old episodic formulation may still have to be preferred.

Examples of Immediate Rewards Let us return to the two toy domains from the previous chapter. Can they be restated in terms of immediate rewards? Yes, they can. In the case of the *maze*, the engineer may decide to reward with $r = 1$ only those actions that reach G in a single step, all other actions receiving $r = 0$. In the variation with potholes, caches, and monsters, specific additional rewards can be provided, say, $r = -2$ for a pothole, $r = -20$ for a monster, and $r = 5$ for a cache. These rewards can even be probabilistic, drawn from distributions with user-specified parameters.

In the *car-races* domain, each step along the track can be rewarded with $r = -1$, any step that reaches the goal with $r = 10$, and any action that results in hitting the wall can be punished by $r = -10$. The choice of the concrete values is in the hands of the programmer who knows that different conception of rewards is likely to result in different learning behavior.

Immediate Rewards and V-Values Let us begin with the simpler scenario that focuses on states, rather than on actions. Suppose that each state, s, has a value, $V(s)$, which is an estimate of the average reward received by the agent between this state and the goal. Suppose that, at time t, the agent is in state s_t and chooses an action that results in state s_{t+1}, receiving immediate reward r_t.

The anticipated reward associated with the state that has been reached at time t is then r_t plus the value of the next state, s_{t+1}. More formally, this estimated reward is $R = r_t + V(s_{t+1})$. Discounting $V(s_{t+1})$ by some coefficient $\gamma \in (0, 1)$, we obtain the following:

$$R = r_t + \gamma V(s_{t+1}) \tag{18.1}$$

Updating V-Values Again, suppose that an action taken at s_t resulted in state s_{t+1}, receiving reward r_t. How shall we update the V-value of s_t based on the reasoning from the previous paragraph? Let us recall the formula recommended for updating V-values in Sect. 17.6:

$$V(s_t) = V(s_t) + \alpha[R - V(s_t)] \tag{18.2}$$

This increases the estimate $V(s_t)$ if it is smaller than R and decreases it if is greater than R; coefficient α then controls the amount of this change. If we use for

R the value obtained from Eq. (18.1), which is $R = r_t + \gamma V(s_{t+1})$, we arrive at the following V-updating formula:

$$V(s_t) = V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \tag{18.3}$$

After some minor rearrangement, this can be reformulated as follows:

$$V(s_t) = (1 - \alpha)V(s_t) + \alpha[r_t + \gamma V(s_{t+1})] \tag{18.4}$$

The last formula is easy to interpret in plain English. The updated value, $V(s_t)$, of state s_t is the weighted average of two terms; the first is the current V-value of the state, and the second is the discounted estimate of the reward received after the given action has been taken and state s_{t+1} reached.

Typical values of the two parameters are $\gamma \in (0.8, 1)$ and $\alpha \in (0.05, 0.15)$.

Temporal Difference: TD(0) Those approaches to *reinforcement learning* that use immediate rewards form a whole family usually referred to by the term, *temporal difference*—essentially the difference between states s_t and s_{t+1} (where t and $t + 1$ are subsequent time instances).

Temporal-difference learning where the modification of the state value is based only the immediate reward, r_t, is sometimes referred to by the acronym $TD(0)$. In Sect. 18.4, we will learn that this is a special case of $TD(\lambda)$ for the extreme case of $\lambda = 0$.

18.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the main weakness of the episodic approach to *reinforcement learning*? Do immediate rewards always correct this weakness?
- How would you specify the immediate rewards in the simple domains used in this textbook?
- Derive the formula that updates the V-value of a given state after an action has been taken. Discuss the impact of its two parameters.

18.2 SARSA and Q-Learning

Having introduced the simple mechanism that updates V-values, let us now proceed to a formula to be used to update Q-values. After this, we will be ready to take a look at the baseline techniques underlying temporal difference.

Updating Q-Values Instead of V-Values Suppose that the engineer wants to update the Q-values of actions and not the V-values of states. Recall that $Q(s_t, a_t)$ is the system's estimate of the average total reward (between s_t and the goal) to be accrued if action a_t is applied to state s_t.

The reasoning behind the following formula is essentially the same as the reasoning that led to the derivation of Eq. (18.3) that updated V-values. There is thus no need to repeat the derivation. Suffice it to say that, once the action has been taken and the immediate reward, r_t, received, the estimate of the action's value is updated as follows:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{18.5}$$

Note that the new value depends not only on the immediate reward but also on the estimated quality of the next state. Further on, let us remind ourselves that α is a user-set constant and γ is a discounting factor.

SARSA Let us assume that the Q-values of all actions have been initialized and that the agent begins at the starting state, $s_0 = S$. At any given state, the agent chooses the next action following the ϵ-greedy or *soft-max* strategy. Having chosen the action, it executes it, receiving reward r_t, and observing the next state, s_{t+1}. The quality, $Q(s_t, a_t)$, of action a_i in state s_t is then updated by Eq. (18.5) from the previous paragraph.

Note that the formula used to update the action's Q-values depends on the quintuple $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$. The letters have given the technique its name: SARSA.

Q-learning The SARSA approach was presented here mainly for its instructional value. Much more popular is its slightly modified version, known as Q-learning. Practically the only difference is the assumption that in the new state s_{t+1}, the next action is selected with the greedy strategy (which always picks the highest-valued action) rather than ϵ-greedy or *soft-max* strategies.

This means that, instead of Eq. (18.5), the following one is used to update the concrete Q-values:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{18.6}$$

Both are summarized by the pseudo-code in Table 18.1.

Domains with Deterministic Rewards The attentive reader will recall that Chap. 17 said that *reinforcement learning* is meant primarily for stochastic domains where the rewards and/or the next states are determined probabilistically. In the simple versions of our toy domains, however, both the rewards and next states could be established deterministically. And indeed, such deterministic domains are often encountered in realistic applications.

Table 18.1 Pseudo-code of SARSA and Q-learning

```
Input: strategy for choosing the action: $\epsilon$-greedy or soft-tmax,
       parameters $\alpha$ and $\gamma$, and parameters $\epsilon$ or $z$ of the employed strategy
       for all state–action pairs, initialized values, $Q_0(s_i, a_j)$, and counts, $k_{ij} = 0$.

1. Set $t = 0$, and determine the initial state, $s_0 = S$.
2. Apply to $s_t$ action $a_t$, selected by the user-specified strategy.
3. Observe the new state, $s_{t+1}$, and the reward, $r_t$.
4. In the case of SARSA, assume that in the new state, $s_{t+1}$, action $a_{t+1}$ is chosen using the user-
   specified strategy. In the case of $Q$-learning, assume that the best action is to be chosen. The
   values are updated as follows:
   SARSA: $Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
   $Q$-learning: $Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
5. If $s = G$, start a new episode by going to 1; otherwise, set $t = t + 1$ and go to 2.
```

If this is the case, the fact that we do not need to estimate average returns leads to a simpler formula for Q-learning:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a) \tag{18.7}$$

18.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Write down the formula that updates Q-values. Discuss its behavior and parameters.
- Describe the principle of the SARSA approach to reinforcement learning. Where did this name come from?
- What constitutes the difference between SARSA and Q-learning?
- Elaborate on the difference between domains with deterministic rewards and domains with non-deterministic rewards. Write down the formulas for both of them, and point out the difference.

18.3 Temporal Difference in Action

It is one thing to master the general principles; it is another to make them work in concrete circumstances. Also, the successful engineer needs to have a good idea about what kind of domains will primarily benefit from the given paradigm.

Main Steps in Implementing the *Car Race* A simple method of implementing the *car-race* problem begins with choosing a grid covering the track and its surroundings. Then a matrix is created whose each field represents one square of the grid and contains the immediate reward received by an agent landing on this square. In a simple implementation, the reward can be $r = -1$ for all squares that find themselves on the track and $r = -10$ for those on the wall and outside the track. The squares on the goal line can be assigned, say, $r = 10$.

In the next step, a lookup table for the states and their actions is created and its Q-values (or V-values) initialized, perhaps by the *optimistic initialization* the reader knows from Chap. 17. The user is then asked to provide the values of the various parameters such as ϵ, α, γ, etc.

Then the program implementing SARSA or Q-learning can be started.

Impact of Rewards The way the rewards are assigned to the actions strongly influences the learning process. Creative rewards will result in fast learning; poor ones can render any learning impossible. The engineer's success will to a great degree depend on the way he or she conveys (by way of rewards) to the program the idea of what should be done and what should be avoided. The flexibility of all available possibilities is much higher than a beginner suspects.

In the *car races*, the rewards should reflect the imaginary driver's goals. In the highly simplified implementation from Sect. 18.1, the immediate rewards depended only on "being on-track" ($r = -1$), "hitting a wall" ($r = -10$), or "reaching the goal" ($r = 10$). Not knowing which direction to choose, the agent that learns from these rewards wanders around aimlessly for quite some time.

Many improvements can be considered. For instance, the agent may receive rewards that are higher if the car moves in the direction of the next curve. Also, the rewards can be reduced if the car comes dangerously close to a barrier, and if it hits the wall, the punishment can be made proportional to the angle of crash. Many other such aspects can be formulated by an engineer knowledgeable of the given application—and most of them will help speed up the learning process.

Another Popular Test-Bed: TSP Consider the *traveling salesman problem* (TSP) from Fig. 18.1. The input is a set of N cities (usually N is much greater than the six shown in the picture) and a table with the distances between all pairs

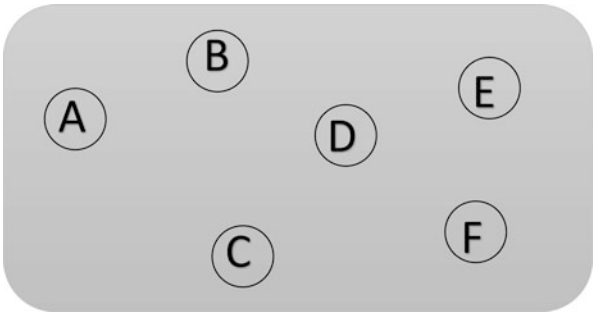


Fig. 18.1 The input of the traveling salesman problem consists of a set of cities with known distances for all pairs. The task is to find the shortest path connecting all cities

of cities. The task is to find the shortest path that connects all cities. Obviously, brute-force number-crunching would have to investigate $N!$ alternative routes, which is impossible for large N. Since many engineering problems can be easily formulated within this framework, the TSP problem is a test-bed of choice for innumerable algorithms such as high-level optimization, search techniques, and swarm intelligence.

Stochastic TSP The traveling salesman, though popular by textbook writers, is still only a simplification of reality. To begin with, the typical goal in textbooks is to minimize the overall distance. In reality, the salesman may want to minimize *time*, a quantity that is determined by distance only up to a certain degree. Other influences, such as weather, road quality, and traffic, surely cannot be ignored. The costs of moving from X to Y is then a random number; worse still, the value of this number may change in time, such as when the traffic is heavy during peak hours and much lighter at night.

There are no limits on the agent's goals and intentions. The agent may also want to minimize fuel, it may decide that it is enough to visit only 90% of the cities, and so on. The number of variations is virtually unlimited.

All these circumstances make the task more attractive for *reinforcement learning* than for many classical approaches, especially for those that rely on deterministic analysis. The *state* is here the agent's location (a city), plus perhaps some information about which cities remain to be visited. The *action* is the decision about where to go next. Finally, the reward is the time it takes to reach the next city.

The reader will find it a good exercise to figure out how to implement the *stochastic TSP* task in terms of the episodic formulation and in terms of temporal-difference learning.

What Constitutes a Good Application for This Paradigm? Car races, mazes, and traveling salesmen have found their way into this text thanks to their instructional values. The world outside introductory instruction is different—though perhaps less so than the pessimistic reader may fear. The engineer's job often consists in finding a way to map the problem at hand onto a problem he or she knows how to handle. While it is unlikely that the readers will ever be asked to implement the traveling salesman for their firm, many concrete problems can be reformulated in terms of the goals and obstacles that the traveling salesman involves.

For a problem to be addressed by *reinforcement learning*, we need to know how to cast it in terms of states and actions and how to allocate the rewards; the rest is creativity. So far, we have been limited by the requirement that all states and actions be discrete and that the lookup table has to be manageable. Later, we will realize that even these limitations can be overcome.

18.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How would you implement the solutions to the classical problem of *mazes* and *car races*? How will the learning efficiency be affected by the way the rewards are assigned?
- Summarize the principle of the *stochastic traveling salesman* problem. How would you address it by *reinforcement learning*?
- What are the typical characteristics of engineering problems that are suitable for *reinforcement learning*?

18.4 Eligibility Traces: TD(λ)

Beyond the baseline versions of SARSA and Q-learning, more sophisticated techniques are frequently used. Although *eligibility traces* fall rather under the "advanced" rubric, let us acquaint the reader at the least with its basic underlying ideas.

One-Step Temporal Difference The first section of this chapter suggested that V-values be updated by $V(s_t) = V(s_t) + \alpha[R - V(s_t)]$. The reward, R, was then estimated by adding the immediate reward, r_t, to the estimated average reward of the follow-up state, $V(s_{t+1})$. Mathematically, this is expressed by $R = r_t + \gamma V(s_{t+1})$. The reader will recall that the following formula was recommended:

$$V(s_t) = V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)]$$

Let us denote the reward calculated in this manner by $R^{(1)} = r_t + \gamma V(s_{t+1})$. The superscript, "(1)," points out that apart from the immediate reward r_t, one additional term is involved: the value of the next state, s_t.

n-Step Temporal Difference The reader already knows that $V(s_{t+1})$ is just an estimate of the average reward received in state s_{t+1}. This estimate can be more accurately expressed as $r_{t+1} + \gamma^2 V(s_{t+2})$, which is the immediate reward at state s_{t+1} plus the estimate of the average reward in the state that follows, $V(s_{t+2})$. Substituting this into $R^{(1)}$, we receive the following:

$$R^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2})$$

There is no reason to stop at this stage. We can just as well continue, applying the principle recursively as long as we wish. In the end, we arrive at the following formula:

$$R^{(n)} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^n V(s_{t+n}) \tag{18.8}$$

Instead of updating with the original $R^{(1)}$, we may update the V-value with $R^{(n)}$:

$$V(s_t) = V(s_t) + \alpha[R^{(n)} - V(s_t)] \tag{18.9}$$

Averaging Out the Rewards $R^{(n)}$ We see that there are many ways to estimate R, each for a different value of the superscript, (n). As every so often, in machine learning, serious trade-offs are involved. The larger number of terms involved in the calculation of $R^{(n)}$ seems to promise higher accuracy; but then, this may be nothing but precise calculations with more numerous crude estimates, hardly a guarantee of success. Not knowing how many terms to use, we may just as well calculate all possibilities and then take the average.

$$R = \frac{R^{(1)} + \ldots R^{(n)}}{n} \tag{18.10}$$

Discounted Averaging Upon reflection, the longer sequences (higher values of n) will be deemed less reliable than shorter ones. It is, therefore, a common practice to average the values with discounts. For this, yet another parameter is used, $\lambda \in [0, 1]$.

$$R_t^{\lambda} = R_t^{(1)} + \lambda R_t^{(2)} + \lambda^2 R_t^{(3)} + \ldots + \lambda^{n-1} R_t^{(n)} \tag{18.11}$$

Let us now make a few minor improvements. First, we will allow n to grow beyond all limits (t ∞). Second, we will express the formula more succinctly with the symbol for summation, Σ. Third, we will normalize all terms by $(1 - \lambda)$ so as to make the values to sum to 1. Putting all of this together, we obtain the following:

$$R_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \tag{18.12}$$

In reality, of course, the number of steps in the episode is finite, and it makes sense to replace in Eq. (18.12) all terms beyond the final step with $\lambda^{n-1} R_t$, where R_t is the episode's final reward. This leads to the following formula (T is the length of the episode):

$$R_t^{\lambda} = (1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} R_t \tag{18.13}$$

This, then, is the reward estimate to be used instead of R in Eq. (18.2) that updates the V-values in this more sophisticated approach.

Eligibility Traces Admittedly, Formula (18.13) is somewhat impractical for a computer implementation. Realistic applications prefer to employ the principle in the form of so-called *eligibility traces*. The idea is to maintain, at each time step t

and each state, s, its eligibility trace $e_t(s)$, which indicates how eligible the given state is for a value update.

At each step made by the agent in state s, the eligibility decays, which means that it is reduced by a factor determined not only by the λ we have already encountered but also by a user-specified $\gamma \in (0, 1]$:

$$e_t(s) = \gamma\lambda e_{t-1}(s) \tag{18.14}$$

A state that has not been visited for a long time has very small eligibility. However, if the state *is* visited, during the agent's random walk, its eligibility is immediately increased by an increment of 1:

$$e_t(s) = e_{t-1}(s) + 1 \tag{18.15}$$

Value Updates Made Dependent on Eligibility Traces Let us introduce the concept of a *TD error* that for each state measures how much mistaken its current value-estimate appears to be. In accordance with what this section has said about state rewards, we will define it by the following equation:

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{18.16}$$

The state's value is then updated by an amount made proportional to the state's TD error and by the "strength" of its eligibility trace. More specifically, the state's value is changed by adding to it the following term:

$$\Delta V_t = \alpha\delta_t e_t(s) \tag{18.17}$$

Here, another parameter, α, has been introduced, fulfilling essentially the role of *learning rate*. Its concrete choice gives the engineer another level of flexibility in controlling the learning process. Typical values will be from interval [0.05, 0.2].

TD(λ) Direct implementation of the eligibility-traces algorithm is somewhat complicated, and common practice therefore relies on the much more practical algorithm TD(λ) whose one episode is summarized in Table 18.2. TD(λ) has been proved to be fully equivalent to the approach described in the previous paragraphs.

Detailed discussion of the whole approach would exceed the scope of an introductory textbook. Suffice it to say that approaches based on eligibility traces often result in faster convergence of the learning process.

TD(λ) has Alternatives: SARSA(λ) and Q(λ) The TD(λ) approach described above carries out gradual updates of V-values; that is, it seeks to estimate average rewards experienced at individual states.

Without going into detail, let us only briefly comment that the approach based on eligibility traces can be applied also to mechanisms to update the Q-values of actions. The names of these techniques are SARSA(λ) and $Q(\lambda)$, and they can be

Table 18.2 One episode of TD(λ)

```
Initialize $t = 0$ and $s_0 = S$.
Before the first episode, the eligibilities of all states are $e_0(s) = 0$.

1. At state $s_t$, choose action $a$ by the $\epsilon$-greedy or soft-tmax strategy.
2. Take action $a$, receive reward $r_t$, and observe the new state, $s_{t+1}$.
3. Increment the state $s_t$'s eligibility: $e_t(s_t) = e_t(s_t) + 1$.
4. Calculate TD error: $\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$.
5. For all states, make the following updates:
   $V_t = V_t + \alpha\delta_t e_t(s)$
   $e_t(s) = \gamma\lambda e_{t-1}(s)$
6. If the goal, $G$, has not been reached, set $s_t = s_{t+1}$ and go to 1.
```

cast as fairly straightforward extensions of TD(λ). Again, both are advanced, and as such beyond the scope of this book.

18.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the term, n-step temporal difference. How do we average a set of temporal differences for different values of n?
- Describe the mechanism that applies n-step temporal difference to updates v-values. What is the basic principle of *eligibility traces*?
- Comment briefly on the advanced techniques known by the acronyms, TD(λ), SARSA(λ), and $Q(\lambda)$.

18.5 Neural Network Replaces the Lookup Table

Up till now, we have assumed that all states, even all actions, can be stored in a lookup table. This is often unrealistic. To begin with, the number of states and actions can be so big (billions or more) that it is impossible to store them all in memory, let alone explore each of them experimentally. Moreover, the states or actions can be drawn from continuous domains, which of course makes it impossible to provide their complete list.

Tic-Tac-Toe Figure 18.2 illustrates the 3-by-3 version of the game. Two players are taking turns, each placing one cross (the first player) or one circle (the second player) on the board. The goal is to achieve a line of three crosses or circles—in a column or in a row or diagonally. Whoever succeeds first wins. Suppose that in

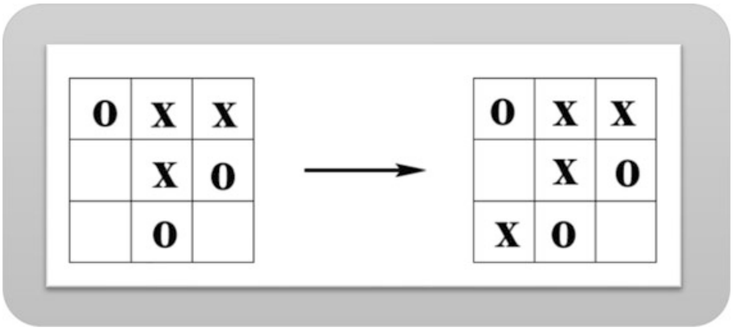


Fig. 18.2 In tic-tac-toe, two players take turns at placing their crosses and circles. The winner is the one who obtains a triplet in a line (vertical, horizontal, or diagonal)

the situation depicted on the left, it is the turn of the "crosses" player; the win is achieved by putting the cross at the bottom left corner. If it was the opponent's turn, he or she would prevent this by placing there her circle.

Even in the very simple 3-by-3 version of the game, the lookup table is going to be very large: each board position represents a state, and the number of states and actions is here dictated by the implacable laws of combinatorics. To wit, the player who begins has nine squares to choose where to place the cross; the opponent can then place the circle at any of the remaining eight squares. In line with this reasoning, we observe that the total number of different actions is upper-bounded by 9 factorial.

While a lookup table of this size can still perhaps be considered, the situation becomes unmanageable as we proceed to, say, a 50-by-50 board. Then we are clearly beyond all reasonable limits, and an alternative solution has to be sought.

Pole Balancing Figure 18.3 illustrates what is known as the *pole balancing* problem. A cart carries a hinge to which is attached a pole. The task is to prevent the pole from falling by shifting the cart in either of the two directions. Each *state* of the game is described by the following four variables: the cart's location and velocity, the pole's angle, and the velocity of the angle's change. As for *actions*, there are only two: (1) apply force in the left–right direction and (2) apply force in the right–left direction. The simplest version assumes that the actions are taken at regular intervals, say, 0.2 s and that the magnitude of the applied force is always the same. The goal is to prevent the pole from falling as long as possible; the reward is proportional to the time before the fall.

The main problem with pole balancing is that the variables it relies on are essentially continuous-valued. True, we can discretize each of them by dividing its domain into small intervals, similarly as we did it in Sect. 2.4. This, however, would result in loosing certain amount of available information, which would lower the chances of reaching a good solution.

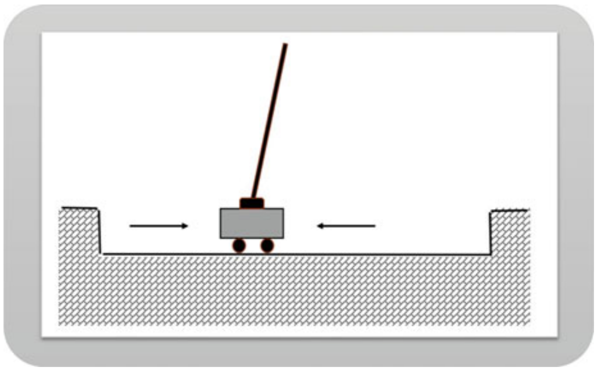


Fig. 18.3 The task is to prevent the pole from falling by sending impulses from left to right or from right to left as needed

States and Actions Rarely Visited The problem with the lookup table's enormous size is not just the difficulties of storing it in the computer memory. If the number of states and actions is extremely high, then even a great number of episodes will fail to visit then all; and many states perhaps *have* been visited, but only so rarely that their V-values and/or Q-values are estimated based on clearly insufficient experience. Put another way, the greater the number of states or actions, the greater the number of experiments needed for reliable estimates of average rewards. Running all these experiments in realistic time may be impossible.

Approximating the Lookup Table by a Neural Network In domains with continuous variables and in domains with too many different states and actions, common practice approximates the task by a neural network.

Quite a few possibilities can be considered. For instance, in some domains, each state permits essentially the same set of actions—as in the *maze* problem where only four actions were possible: *up, down, left*, and *right*. In this event one may consider a scenario where the input of the network represents a state (described by a vector of binary attributes), and the output represents actions. To be more specific, each output neuron represents one action, and the action whose neuron outputs the highest signal is then taken. For neural activation functions, either ReLU or LReLU is typically used (rather than `sigmoid`).

Another implementation possibility assumes that the (binary) input vector has two parts: one to represent the state and the other for the action. The output of the network then gives the value of the action in the given state. In this case, the backpropagated error is usually based on the *loss* function. For instance, one can backpropagate $\log_2 p_i$, where p_i is the output of the i-th neuron when i-th action is the correct one.[1]

[1]For activation functions and the backpropagated loss function, see Chap. 6.

18.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Provide an example of an application domain where the number of states and actions prevents us from using a lookup table.
- How can the situation be handled by a neural network? What will be the network's input and what is its output?

18.6 Reinforcement Learning in Game Playing

Let us illustrate the use of a neural network in *reinforcement learning* using a famous application that, in the 1990s, helped draw the attention of the scientific community to the vast potential of these techniques.

Why Backgammon? The reader is sure to have heard of the game of *Backgammon* whose one possible state is shown in Fig. 18.4. The reason the game is important here is that it represents a really challenging test-bed. The number of states is very high, in each state the player has the choice of several different moves, the next state depends on the selected move only to a limited extent, and at each moment a lot depends on the outcome of the rolled dice. For a long time, the game was considered to be almost unsolvable for classical artificial intelligence. Even the idea of *reinforcement learning* based on a lookup table appeared hopeless.

Then, in the late 1990s, the astonished world was informed that a machine-learning program did beat the world champion, after all. The secret was to replace the lookup table with a neural network.

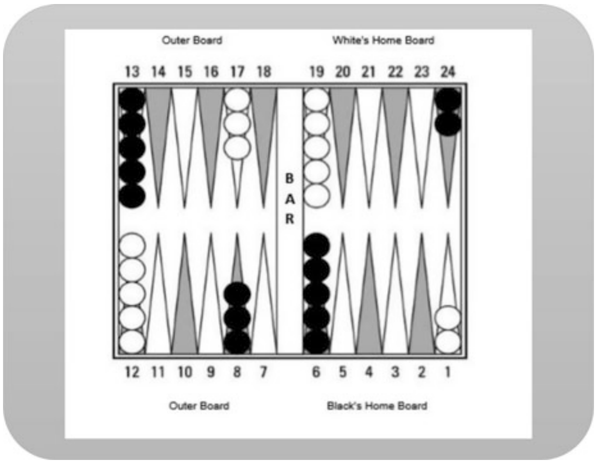


Fig. 18.4 Backgammon: an application where *reinforcement learning* needs to be helped by neural networks

Backgammon's Rules (Heavily Simplified) Let us briefly outline the essence of the game's rules. The board has 24 locations called *points*. Two players are taking turns. One has 15 white pieces playing counterclockwise, starting from bottom left; the other has 15 black pieces playing clockwise, starting from upper left. Each player seeks to get all his or her pieces around the whole board: the white wants them to end up in the upper-left corner, and the black them in the bottom left corner.

Rolling two dice, each player learns how far his or her pieces can be moved. For instance, one piece can be moved by two points and the other by four. Usually, quite a few alternative possibilities exist as to which pieces to move and how much. The concrete choice is limited by the constraint that one is not allowed to land a piece of a point that is already occupied by two of the opponent's pieces. Experienced players are adept at "speeding up" the movements of their pieces while impeding those of the opponent.

The rules are more complicated than this, but further details are unimportant, for the rest of the section.

Main Difficulty: Too Many States Obviously, the number of states in this game is very high, dictated by what mathematicians sometimes call combinatorial explosion. Moreover, in each state the player is to choose from quite a few actions. The situation of anyone who wants a computer to learn to play this game is further complicated by the randomness implied by the rolled dice.

Neural Network for *Backgammon* The simple network in Fig. 18.5 was used by a famous *reinforcement learning* program that proved capable of learning to play the game at the master level and beyond. Let us take a closer look.

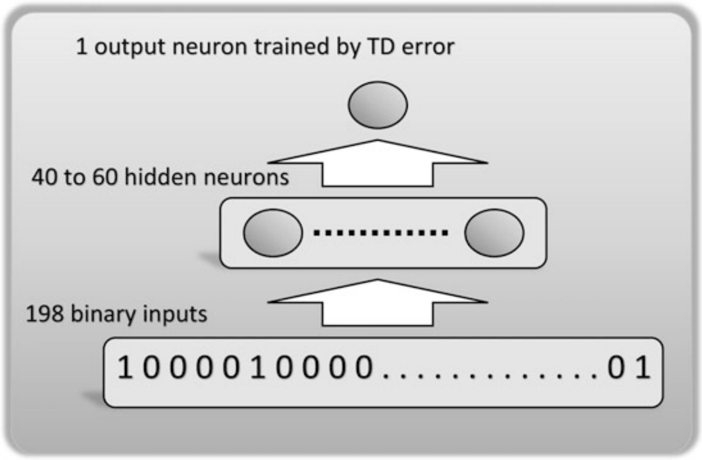


Fig. 18.5 Backgammon: a neural network that plays the role of the lookup table is trained by the TD error

The input is a feature vector that describes the state of the game. As mentioned earlier, the board has 24 points, each of which is either empty or occupied by a certain number of white and/or black pieces. In the game's early implementation, the *one-hot* representation was used, describing each point by eight bits, four for white and four for black. If the point is occupied by no piece, all bits are set to zero. If it contains, say, three white pieces and no black pieces, then the third "white" bit is set to one, and all other bits are set to zero. Apart from these $24 \times 8 = 192$ bits, the network has additional six inputs that represent certain special aspects that the above summary of the rules did not mention. All in all, the 198 inputs are capable of capturing any state of the game.

There is only one output neuron that for each state presented at the input layer returns its V-value (for more about this, see the next paragraph). The network has only one hidden layer containing a few dozen neurons.

State-Value Updates and *TD Errors* The attentive reader knows that, in the simplest implementation, the V-values are updated by the following formula:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha[r_t + \gamma V_t(s_{t+1}) - V_t(s_t)] \tag{18.18}$$

The difference between the value at time t and the updated value at time $(t + 1)$ is the *TD error*; it serves as the network's error to be backpropagated after the presentation of state, s_t, and after the calculation of its V-value update.

$$E_{TD}(s_t) = V_{t+1}(s_t) - V_t(s_t) \tag{18.19}$$

Eligible Traces Were Used Even the simple network described above was capable of achieving surprising playing strength. To make it beat the strongest human masters, however, additional "tricks" were needed. Among these, we can mention the somewhat more advanced training approach based on the *eligibility traces* whose principle was outlined in Sect. 18.4.

18.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Summarize the main difficulties faced by an engineer seeking to implement a Backgammon-learning program.
- Explain how the individual states can be described by a feature vector.
- Explain the principle of the *TD error* that was used during this network's training.

18.7 Deep-Q-Learning

In many games, each state can be represented by a two-dimensional matrix. The winning chances of either player may then be indicated by tell-tale patterns that can be discovered by neural networks, especially when *deep learning* has been used for their training (see Chap. 16).

Another Test-Bed: GO Many readers will be acquainted with the game known as GO whose one possible state is shown in Fig. 18.6. For our needs, the detailed rules are unimportant. Suffice it to say that two players take turns, one with white stones and the other with black; they try to place their stones on the board in a manner that satisfies specific goals such as surrounding (and thus destroying) the opponent's pieces. The game is known to be so difficult that the field of artificial intelligence for many years failed to develop a computer program that might compete with strong amateur players, let alone masters.

Too Many States and Actions On the 19-by-19 board, the first player places his or her stone on one out of the $19 \times 19 = 361$ fields. Then the opponent places a stone on one of the remaining 360 fields, and so on. The numbers of states and actions are here so enormous that a lookup table cannot even be considered.

Deep-Q-Learning to Discover Meaningful Patterns After long practice and deep studies, strong human players become adept at seeing on the board specific patterns that indicate which next move to consider. A glance at Fig. 18.6 will convince us that these patterns are non-trivial and probably difficult to discover. Can they be learned by a neural network?

Advanced visual patterns are typical of many other games, notably *Atari*-games where perhaps the oldest attempts to employ deep learning were made. One way to address the situation is to treat each 2-dimensional game-state as one computer-

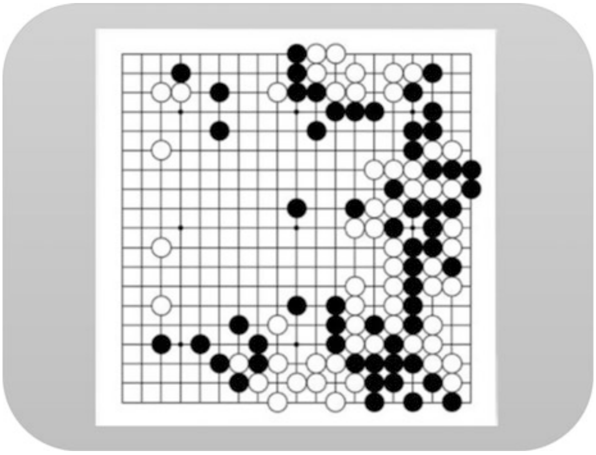


Fig. 18.6 Each state in GO can be understood in terms of visual patterns. This observation leads to the idea to train a GO-playing neural network by *deep learning*

vision image to be input into a convolutional neural network such as those we met in Chap. 16.

Backpropagated Error in Deep-Q-Learning Suppose that an action a_t is taken in state s_t, which results in state s_{t+1} and immediate reward $R(a_t, s_t)$. The reader already knows that the quality of an action taken in a given state is the estimate of its average reward, $Q(a_t, s_t)$.

Let the highest-reward action at state s_{t_1} be a_{t+1}. Here is the estimated reward of a_t in s_t as employed in Q-learning (note that the quality of the next state is discounted by γ):

$$R(a_t, s_t) + \gamma \max_{a_{t+1}} Q(a_{t+1}, s_{t+1})$$

This means that the mean squared error to be backpropagated through the network is given by the following formula:

$$MSE = Q(a_t, s_t) - [R(a_t, s_t) + \gamma \max_{a_{t+1}} Q(a_{t+1}, s_{t+1})] \tag{18.20}$$

Network's Architecture As always in neural networks, the engineer's decision about the concrete size and architecture can significantly affect the software's performance. The network's complexity has to reflect the difficulty of the problem at hand: neither too simple nor too big, that is the golden rule. We must not forget that the more trainable parameters are involved, the larger training set is needed.

A lot of alternative solutions exist, offering a broad scope for creativity. For example, one can represent the problem with a single matrix, but one can also consider two cooperating matrices, one for white pieces and the other for black. Convolution neural networks known from Chap. 16 are commonly employed here.

Famous Success Story It was something of a sensation when, in 2016, a program called *AlphaGo* succeeded in beating the human world champion. The software used two neural networks, one to choose the move and the other to predict the winner. All in all, millions of synaptic weights had to be trained. In an early stage, the program learned from numerous amateur games, later continuing the training by playing against multiple versions of itself. Reinforcement learning helped it gradually to improve until it reached the level of the strongest human players.

The very next year, 2017, an even stronger program was introduced under the name of *AlphaGo Zero*. This was reported to have learned exclusively from playing against itself. When doing so, it started with a totally random behavior and gradually improved. After just a few days' training, the program was strong enough to beat even the previous program, *AlphaGo*. It clearly outperformed many centuries of human experience, sometimes even discovering previously unknown strategies.

Later, the same learning principle was applied with similar success to other games such as shogi or chess, easily reaching a world-championship level in all of them. A genuine general-purpose learning system was born.

Optimistic Conclusion Until recently, many a domain was seen as an inviolable domain of humans, totally beyond the reach of machine intelligence, which largely relies on number-crunching power. From this perspective, the existence of programs that beat human champions in Backgammon and GO is encouraging.

There is no doubt that *reinforcement learning*, coupled with *deep learning*, can enhance human ingenuity well beyond yesterday's dreams.

18.7.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Under what circumstance do we resort to deep-Q-learning?
- Comment on the architecture of the networks used in deep-Q-learning. What error value is here typically backpropagated?
- Summarize the lessons from *AlphaGo Zero*.

18.8 Summary and Historical Remarks

- In many applications, the episodic approach from the previous chapter appears somewhat cumbersome, which is why the more efficient *temporal difference* is often preferred. The point is not to wait till the end of an episode, but rather learn from immediate rewards.
- The most popular algorithms in *temporal difference* are SARSA and Q-learning. Both focus on updating the Q-values of the actions. In domains with too many actions, one can consider updating only the states' V-values.
- When updating the V-values, the baseline *temporal difference* looks only one step ahead. The more sophisticated approach of TD(λ) looks deeper, relying on the so-called *eligibility traces*. Analogous approach is employed by SARSA(λ) and Q(λ) that update the Q-values.
- The simplest versions of the techniques rely on lookup tables. Their practical application was here illustrated on such test-beds as labyrinths, car races, and the stochastic traveling salesman.
- In some domains, lookup tables are impractical. For one thing, the number of different states and/or actions can be prohibitively high; for another, the states and actions are described by continuous-valued variables which cannot easily be represented by the rows in the lookup table. In these domains, the lookup table is often approximated by a trainable neural network.
- The potential of *reinforcement learning* to master even very difficult tasks was demonstrated by computer programs that learned to play such games as Backgammon and Go at the level of the strongest human players.

- In some applications, the states can be described by two-dimensional matrices. This is the case of such games as Atari or Go. In this event, the *deep learning* techniques from Chap. 16 can be used to train the agent. One such approach is called *deep-Q-learning*.

Historical Remarks The principle of TD(0) was introduced by Sutton (1988). The slightly more sophisticated SARSA was invented by Rummery and Niranjan (1994), although the name itself came into use a few years later. The first journal publication of Q-learning (including eligibility traces and Q(λ)) was by Watkins and Dayan (1992), but the origin of these ideas seems to be from Watkins's PhD dissertation in 1989. The advent of deep learning soon led to investigation of the possibilities it can offer to reinforcement learning. Here, Mnih et al. (2015) deserve to be mentioned. AlphaGo was developed in the same year by the company DeepMind.

18.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

18.9.1 Exercises

1. Calculate the number of state–action pairs in the tic-tac-toe example in Fig. 18.2. How many states exist in the case of a 4-by-4 board?
2. Consider the simple maze from Fig. 17.2. Suppose that the values of all states were initialized to $V(s_i) = 0$. Suppose, further, that the immediate reward is 1 for the goal state and 0 for any other state. How will the V-values of the individual state be updated during the agent's first random walk? For value updates, use the formulas from Sect. 18.1.
3. Repeat the previous exercise using SARSA and Q-learning. For value updates, use the formulas from Sect. 18.2.

18.9.2 Give It Some Thought

1. How many episodes might be needed to solve the simple version of the tic-tac-toe game shown in Fig. 8.7?

2. Suggest a few alternative ways of generating rewards of the car-racing domain when addressed by temporal difference. How will each of them affect learnability? What way of assigning rewards will make the learning faster?
3. Consider a difficult game of your own choice, perhaps chess or Go. How would you represent the individual states of this game? What data structures would you use?
4. There is a small difference in the formulas used for Q-value updates in SARSA and Q-learning. Under what circumstance will the first be more appropriate than the second, and under what circumstances will the second be more appropriate than the first?
5. Section 18.4 introduced the idea of eligibility traces. What do you think about the practical merits of this approach? Under what circumstances will it improve learning, and under what circumstances will it do more harm than good?

18.9.3 Computer Assignments

1. Write a computer program that implements the principle of temporal difference. Use this program to solve either the labyrinth or the car-racing problem from this chapter.
2. Design an experiment that will compare the learning performance of *temporal-difference* learning with that of episodic learning: choose an appropriate test-bed, and experiment with different values of various parameters.
3. Consider the map of a big city. Starting at some point X, you need to reach point Y. Typically, many different routes can be taken, each consisting of segments of different lengths, traffic densities, and all sorts of obstacles including traffic lights, potholes, and traffic accidents. You want to find the route that minimizes time and fuel consumption.

   How would you address this problem by the techniques from this chapter? Design appropriate data structures, write the program (similar to the one from the previous task), decide about the best way of assigning rewards, and then run the learning algorithm.

Chapter 19
Temporal Learning

In classical concept learning, each training example is a vector of attribute values. At a more advanced level, however, each example can be a series of such vectors. For instance, one such series can represent the sine function and another may represent an exponential function, and we want the computer to tell them apart. Alternatively, the computer may be asked to alert the user that a pulse from source A was immediately followed by a small growth in signal from source B. In both cases, the goal is to recognize specific temporal patterns. This is why we need techniques to enable *temporal learning*.

The two fundamental goals can be illustrated by the music metaphor where a tune is understood as a series of signals. One possible task is to recognize a concrete song; the other is to complete the tune upon hearing the first few bars.

The chapter first describes early attempts to address these goals by shift registers and multilayer perceptrons. Then it proceeds to the later conception of *recurrent neural networks*. Finally, the basic principles underlying the currently popular mechanism of *long short-term memory* are introduced.

19.1 Temporal Signals and Shift Registers

Let us explain the two fundamental goals of temporal learning. Once this is understood, we can proceed to some simple ways of dealing with them by traditional machine learning.

Two Tasks Figure 19.1 illustrates the task of *temporal learning*. In the picture on the left, each note is an example, but what interests us is their time-ordered series because a tune is a sequence of notes. Domains of this kind present machine learning with two basic tasks. The first is to train a classifier so that it recognizes *what* tune this is. The second is to make sure that the classifier is able to complete the rest of the tune as far as it "remembers" it. Here are the two goals:

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_19

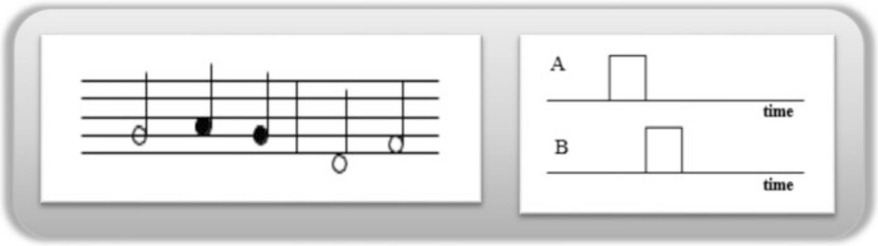


Fig. 19.1 Left: the two fundamental tasks can be characterized as *recognize the tune* and *complete the tune*. Right: the pattern is "pulse B follows pulse A"

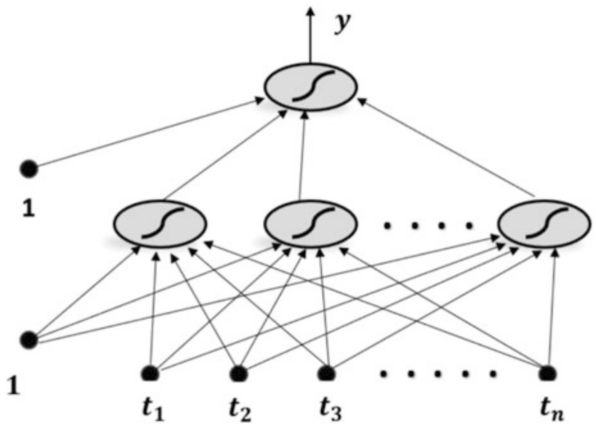


Fig. 19.2 Each input of the neural network represents one time sample

1. *Task 1*: recognize the tune.
2. *Task 2*: complete the tune.

The right-hand part of Fig. 19.1 shows a more technical application where "pulse B follows pulse A." In domains of this kind, too, we either want to recognize a pattern or complete the pattern.

Shift Registers A simple solution relies on what is somewhat inaccurately known as *shift registers*. The principle is shown in Fig. 19.2 for the case of a single time-ordered signal whose samples are presented at the input of a multilayer perceptron. Here, the symbol t_i refers to the i-th time instance. The reader has noticed that the oldest sample (t_1) is presented on the left, and the latest sample (t_n) is presented on the right.

Usually, the sequence of input signals is longer (perhaps *much* longer) than the number of the MLP's inputs. Of this sequence, only the first n signals are presented at the beginning of the training session. For these n signals, the target value is presented at the network's output as indicated in Fig. 19.2.

Once the network's output has been compared with the target, and the error backpropagated to the input, the sample of the input sequence is shifted to the right, a new target value presented, and the error backpropagated. This is repeated until the entire sequence has been presented.

Example Application: Reading English Text Pronunciation of a letter in an English text depends on its context: on what precedes it and what follows. This is why the input of the MLP in Fig. 19.3 consists of a 5-letter excerpt (note that a space, too, is treated as a letter). Each output neuron represents one of the dozens of English phonemes. The target vector consists of many 0s, and a singe 1[1] whose location tells us which phoneme corresponds to the letter in the middle of the input (in this case, this is letter "s.").

A training example is represented by the presentation of one 5-letter sample, together with the correct phoneme at the output. Next, the text is shifted by one letter to the right so that "I" disappears from the network's input on the left, and instead a space appears on the right. The letter to be pronounced will now be "e." In this manner, the entire text is presented, one shift at a time, completing one epoch. As usual, many presentations of the same text (many epochs) will be needed.

Figure 19.3 is only a simplification. In reality, more than one input will be needed for each letter—for instance, its 16-bit definition in the ASCII code: in this case, the five letters will require an 80-bit input. Proceeding to the next letter then means shifting the input by 16 bits.

How Long Is the Context? As always, success will depend on a number of parameters: the size of the hidden layer(s), the learning rate, and the choice of the activation function. However, more important from the perspective of this section are parameters related to the temporal context. Thus in the English-pronouncing domain, the user has to specify the number of letters presented at the input—the five in the above example may not be the best choice.

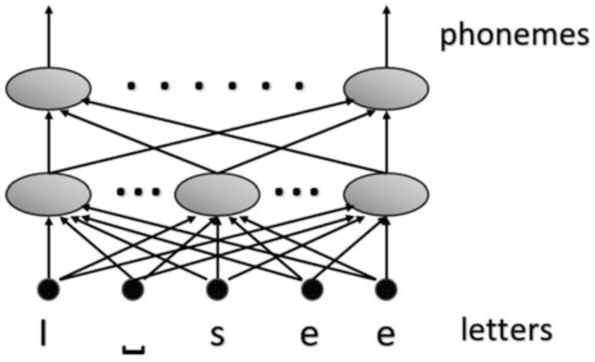


Fig. 19.3 Learning English pronunciation. Each input is a letter, each output a phoneme. The text shifts from right to left, one letter at a time

[1]This is what previous chapters called *one-hot* representation.

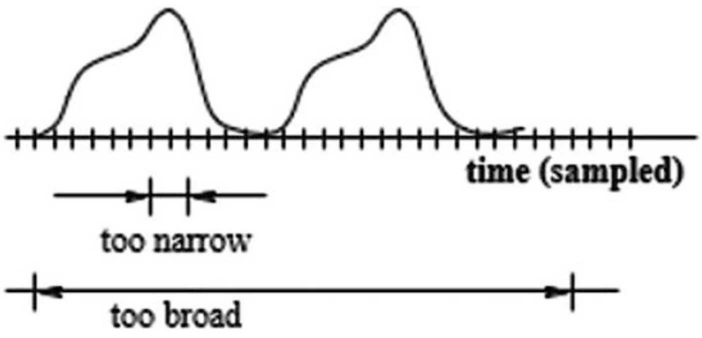


Fig. 19.4 One difficulty with the shift-registers approach is that the engineer may not be able to specify the ideal length of the sampling interval

If the input is an analog signal, then the engineer has to specify the sampling frequency. For one possible complication, see Fig. 19.4. Here, the task is to recognize an electrical signal of a specific shape, and the network is presented with discrete-time samples of this signal. Here is the problem: if samples are rare, they may not capture the signal's essence; if they are dense, then the substance can be lost in excessive details.

Another Shortcoming Presenting the subsequent signals at the MLP's input can confuse the *absolute position* and the *relative position*. The following four-bit inputs represent a specific pattern (a pulse surrounded by zeros), each time shifted by one bit:

- 1st input: 0001
- 2nd input: 0010
- 3rd input: 0100
- 4th input: 1000

Although the pattern is always the same, only shifted by one bit at a time, the network's input is different after each shift.

19.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of *shift registers* using the "pronunciation" domain. What are the inputs and what are the outputs? What is actually being shifted?
- What parameter values does the user have to specify? What are the limitations of this approach?

19.2 Recurrent Neural Networks

Much more flexible than the shift registers are *recurrent networks* where the input receives some of the previous output or internal signals. Two baseline architectures are shown in Fig. 19.5.

Bringing Back the Previous Output The intention behind recurrent neural networks is to offer a mechanism for implicit representation of time. In the upper part of Fig. 19.5, this is accomplished by adding to the input also the network's output from the previous time step. Consequently, the network reacts not only to the current feature vector but also to its own previous output. Practical experience shows that this architecture is good if we focus on the *complete the tune* task defined in the previous section.

Here is why it works. The input obtains information about the current input as well as about the network's response to the previous input. That one, in turn, also contained, implicitly, the network's reaction to even older inputs, and so on. At least in theory, then, the memory might be very long.

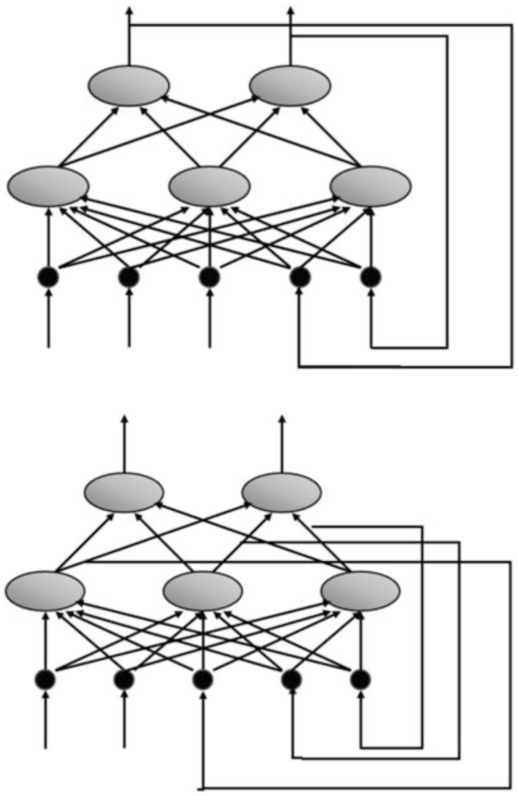


Fig. 19.5 Two basic implementations of recurrent network. In the first, the output signal is brought back to the input, in the second, it is the hidden-layer signal that is brought back to the input

The *recurrent* links can have weights that can be trained just as in the classical MLP; however, this is not necessary, and the network can still do a good job if all the recurrent weights are permanently fixed to some small number such as 0.1.

Bringing Back the Previous Internal State Sometimes it is more reasonable not to add to the input the previous outputs, but rather the network's previous internal state; this means to bring back the outputs of the hidden layer(s) as indicated in the bottom part of Fig. 19.5, Again, the recurrent links may or may not have trainable weights.

Practical experience shows that this architecture is good if our goal is to *recognize the tune* as defined in the previous section. Again, the thinking is that the network can remember even a very distant memory.

Being Flexible The two architectures from Fig. 19.5 can be combined: the network's input then consists of three groups of signals: (1) the current example's attributes, (2) the network's previous output, and (3) the network's hidden-layer's previous output.

This arrangement combines the advantage of both approaches, and practical experience confirms that it can be used for both basic goals of reinforcement learning. After all, to be able to complete the tune, we must first recognize it. However, the number of inputs and the number of trainable weights can then become impractically high.

Criticism When they first appeared, *recurrent networks* were greeted with a lot of optimism. However, practical experience soon revealed that MLPs with recurrent links could learn only short and simple patterns; long and complicated patterns resisted. Of course, there was always the expectation that difficulties would be overcome by an increased number of hidden neurons, and by using more hidden layers, but this was only in theory. Practically speaking, a larger network means more trainable weights, which, in turn necessitates larger training sets and many more epochs—and, in the end, prohibitive computational costs. To conclude, recurrent neural networks turned out to be capable of handling only relatively simple tasks.

Today's computers are much faster, but our ambitions have grown, too. A task that was deemed advanced a generation ago, is from today's perspective regarded as quite simple. These techniques are no longer viewed as panacea capable of addressing any major problem in temporal domains.

Perhaps the heaviest blow was the advent of deep learning, and of the *long short-term memory* that will be introduced in the next section.

19.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Draw the pictures of the two recurrent neural networks introduced in this section, and explain how they are used.
- Discuss the strengths and shortcomings of the two architectures of recurrent neural networks from this section.

19.3 Long Short-Term Memory

When they were first introduced, the techniques from the previous sections were greeted with applause and anticipation. Soon, practical experience dampened the enthusiasm; results were often disappointing. All theoretical arguments notwithstanding, recurrent neural networks' ability to remember a longer past was limited.

Fortunately, a more powerful technology was soon introduced: the *long short-term memory*, LSTM. Its success is probably to be attributed to the ingenious way in which it combines the two older approaches: shift registers, and recurrent network.

Recurrent Neuron The basic idea is something we already know: a neuron whose output is carried back to its input. This can be implemented in many different ways. Either the neuron's own single output is presented at the input, all it is joined by the outputs of some or all of the other neurons in the same layer—as was the case of the network in the bottom part of Fig. 19.5. But whatever the variation, the principle remains the same: the *recurrent neuron*.

Activation Function Perhaps the most common activation function in LSTMs is the same `sigmoid` that was so popular in the first-generation multilayer perceptrons.

Essence of LSTM Figure 19.6 shows the LSTM's basic principle. Similarly as in the case of shift registers, the different time samples are presented at the input layer side by side. The reader will recall the example from Fig. 19.3 where the text "I see" was presented so that "I" was on the left, and the last "e" was on the right.

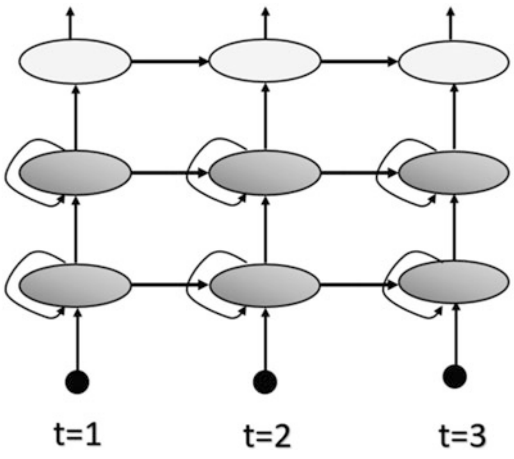


Fig. 19.6 LSTM combines recurrent neurons with shift registers

These inputs are then passed on to the hidden layers of recurrent neurons in the same manner that we saw in in Chap. 6. The picture shows two hidden layers; in reality, there can be only one, but there can also be three or more, the number of these layers being a user-specified parameter.

Note the links emanating from the hidden-layer neurons. The reader can see that the neuron’s output is not only passed on to the next layer but also brought back to the neurons’ inputs just like in classical recurrent networks. What is different, though, are the ‘horizontal” arrows that indicate that the output is sent also to the nearest neighbor to the right. This is perhaps the main difference between LSTMs and MLPs.

The backpropagation-of-error training than follows the opposite direction than the one indicated by the arrows. Note that this means that also the weights of the “horizontal” links are trained.

Why It Works Previous paragraphs, as well as Fig. 19.6, make it clear that the memory of older states is here preserved in a more systematic way than in the case of plain recurrent networks.

Variations The number of variations on this theme is almost inexhaustible. As always, in neural networks, the engineer can consider different numbers of hidden layers and their different sizes. There can be many input signals or just a few, meaning that instead of the five letters in the “I see” example, a longer excerpt can be used. In the case of continuous input signals (recall Fig. 19.4), diverse sampling frequencies can be used. The recurrent weights can be trainable or fixed.

19.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of the *recurrent neuron*.
- Draw a picture of the generic architecture employed by the long short-term memory. Explain how the signals are propagated through the network.
- Briefly discuss possible variations on this theme.

19.4 Summary and Historical Remarks

- One task that the previous chapters largely ignored is the need to identify a series of input signals. The two generic tasks can be characterized as “recognize the tune” and “complete the tune. Other possibilities include the requirement that a difference between two input signals be correctly identified.
- The oldest approach to temporal learning was inspired by the shift registers known from electrical engineering. The idea was to present samples of either

the entire signal or of its portion at the MLP's input so that the oldest sample is on the left and the newest sample is in the right.

- Another solution is known as the *recurrent neural network*. Two approaches were presented here. One of them brings back to the MLP's input the network's previous output. The other brings back the previous state of the hidden layer. The two alternatives can be combined in a single network.
- Perhaps the most popular approach to induction of temporal patterns is the principle of *long short-term memory,* LSTM, that can be seen as a combination of the idea of recurrent networks with the idea of shift registers. The basic unit of LSTM is the *recurrent neuron*.
- The principle of LSTM allows almost inexhaustible range of variations. In this introductory text, only the basic ideas could be presented.

Historical Remarks The first attempts to address temporal learning by neural networks with shift registers goes back all the way to the legendary book by Rumelhart and McClelland (1986). Recurrent networks were studied just a bit later, the oldest paper seems to be Elman (1990). The principle of long short term memory was introduced by Hochreiter and Schmidhuber (1997).

19.5 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

19.5.1 Exercises

1. Chapter 6 once mentioned that artificial neural networks can be seen as visualizations of specific mathematical formulas. More specifically, Eq. 6.4 expressed the network's output as a function of its inputs. Can you write a similar formula for the recurrent neuron?
2. Consider a recurrent neuron that has only one input signal (plus the recurrent output value). Let the activation function be the classical `sigmoid`. What will be the sequence of this neuron's outputs if the input is the following sequence: $(0.1, 0.9, 0.1)$?

19.5.2 Give It Some Thought

1. Figure 19.5 showed two competing versions of the *recurrent neural network*; one of them bringing back the output signals of the previous example, the other the

hidden-neurons' outputs. What difference in behavior does this apparently small difference result in?

2. What exactly is the source of the added flexibility of the *long short-term memory* networks? Why do they outperform classical *recurrent neural networks*?
3. How would you combine LSTM with *deep learning*? Can you imagine a possible application domain where such combination would be needed?

19.5.3 Computer Assignments

1. Implement the two versions of *recurrent neural network* from Fig. 19.5. Make sure your program has an input parameter that tells it which of the two versions it should implement, and perhaps even allowing the combination of the two.
2. Implement a simple version of the *long short-term memory*. User-specified parameters include the number of inputs, and the number of (recurrent) hidden neurons. Create a simple training set serving as a test-best on which to evaluate your program.
3. As a potential test-bed, create a series of values representing concrete functions. A good candidate is $\sin(ax)$ for different values of coefficient a. Can any of the programs you implemented in the previous two questions be trained to distinguish between $\sin(x)$ and $\sin(2x)$?

Chapter 20
Hidden Markov Models

Some temporal patterns are difficult to detect, and to learn, because they are *hidden*: only indirect clues are telling us what is going on under the surface. Problems of this kind fall under the rubric of *Hidden Markov Models*, HMM.

Such a model is defined by a set of states and a set of observations. The agent does not have direct knowledge of the system's state, but it can guess it from the known probabilities of the state-to-state transitions, and from the probabilities of specific observations being made in individual states. Using the right probabilistic formulas, the agent can calculate the likelihoods of concrete sequences of states and/or observations under diverse circumstances. The task for machine learning is to estimate all these useful probabilities from available training data.

The chapter begins with the much simpler *Markov processes* and uses them to develop some initial understanding of how to handle the requisite calculations. Building on these preliminaries, the remaining text explains the principles of the real *Hidden Markov Models*. The three fundamental problems of HMM are presented, together with their solutions, and some typical applications are described.

20.1 Markov Processes

Let us take a look at the so-called *Markov processes* that represent highly simplified version of what this chapter is all about. Once we get used to the mechanism of the calculations involved in this much simpler paradigm, the transition to *Hidden Markov Models* will be quite natural.

Informal Definition Consider a series of time instances, $t_1, t_2, \ldots, t_i, \ldots$, and suppose that a system always finds itself at one out of two states: `hot` and `cool`. Between any two times, t_i and t_{t+1}, the system can stay in its current state, or transfers to the other state. The probabilities of these transitions are known. For instance, we may be told that between t_i and t_{t+1}, the state will change from `hot` to `cold` with 30% probability and that it stays `hot` with probability 70%.

Such a series of transitions is called a *Markov process* if the probability of the system's concrete state at time t_{t+1} depends only on the system's state at the time immediately preceding, t_i, and not on any older history.

Numeric Example Suppose the system knows only two states, `hot` and `cold`, and let the probabilities of state transitions be specified by the following table:

	`hot`	`cold`
`hot`	0.7	0.3
`cold`	0.2	0.8

The entries provide conditional probabilities. For instance, the probability of the state becoming `cold` after having been `hot` is $P(\texttt{cold}|\texttt{hot}) = 0.3$. In each row, the probabilities have to sum to 1 because the system either stays in its previous state or changes into the new state; there is no other possibility. In this particular example, only two states are considered, but the numbers in each row sum up to one even in systems with more than just two states.

Matrix of *Transitions*, A Let us follow the formalism preferred by the field of Hidden Markov Models, even though we are still dealing only with Markov processes (where nothing is "hidden"). The previous table is captured by a matrix A where rows represent the states at t_i and columns represent the states at t_{i+1}:

$$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix}$$

Again, this means that the probability that the `hot` state (first row) will change into the `cold` state (second column) is 0.3, which means 30%. Note that the first row and the first column represent `hot`; and that the second row and the second column represent `cold`.

Vector of *Initial States,* π History has to begin somewhere, at some initial state at time t_0. In our example, the initial state is either `hot` or `cold`, each with its own probability. The vector of these probabilities is denoted by π. For instance,

$$\pi = [0.4 \quad 0.6]$$

This particular vector is telling us that the probability of the initial state being `hot` is 40% and the probability of the initial state being `cold` is 60%.

20.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- How is a *Markov process* defined?
- What do we understand under the term, *transition probabilities* and in what way are these probabilities presented?
- What do we mean by the *probabilities of initial states*?

20.2 Revision: Probabilistic Calculations

Throughout the rest of this chapter, certain types of calculations will keep returning. They are based on probabilistic notions that are rather elementary, but there is no harm in revising them here.

Mutually Exclusive Events A die has been rolled. What is the probability that it gives less than three points? This is easy to establish. The requirement of `less-than-3` is satisfied by two different outcomes: `1-point` and `2-points`, each of them with probability of 1/6. Therefore, the probability of `less-than-3` is $1/6 + 1/6 = 1/3$.

The example illustrates an important concept from the theory of probability. The outcomes `1-point` and `2-points` are *mutually exclusive* because they can never occur at the same time. As we saw, the probability of either of them occurring is the sum of their individual probabilities.[1]

Independent Events Suppose the die is rolled again. The outcome of this second trial clearly does not depend on the previous trial. In this sense, the two events are *independent*: the outcome of one does not depend on the outcome of the other.

The probabilities of independent events are obtained by multiplication. In the specific case of rolling a die, the probability of both trials resulting in `1-point` is $(1/6) \cdot (1/6) = 1/36$.

Example Revisited Previous section relied on an example system that can at each moment be only in one out of two states, `hot` and `cold`. To generalize a bit, let us denote the two states by X and Y.

The reader will recall that matrix A specified the probabilities of state transitions, and that vector π specified the probabilities of initial states.

$$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix} \qquad \pi = [0.4,\ 0.6]$$

[1]By contrast, here is an example of two events that are *not* mutually exclusive: `odd` and `less-than-4`. For instance, 3 is both odd and smaller than 4.

In the matrix, the first row and the first column represent X; the second row and the column represent Y. In π, the first element is the initial probability of X; the second element is the initial probability of Y.

Probability of $s_0 = X$ and $s_1 = Y$ Let the initial state (at time t_0) be denoted by s_0; and let the state at time t_1 be denoted by s_1. Suppose that $s_0 = X$, which then changes to $s_1 = Y$.

The two events, $P(s_0 = X)$ and $P(s_1 = Y|s_0 = X)$, are mutually independent. The probability of these two occurring at the same time is therefore obtained by multiplication:

$$P(s_0 = X) \cdot P(s_1 = Y|s_0 = X)$$

Probability of a Concrete State at Time t_1 In the previous paragraph, we knew that the initial state was $s_0 = X$. Suppose, however, that we do not know what the initial state was. What can we say about the probability of state $s_1 = Y$ in this situation? Put more formally, we want to know the probability $P(s_1 = x)$ that is not conditioned on any previous state.

State X at time t_1 can result from two mutually exclusive situations. The first is that the initial state is X and the system stays in X; the second is that the initial state is Y and the system changes to X. These two cases being mutually exclusive, the resulting probability is calculated as the sum of the two contributing probabilities:

$$\begin{aligned} P(s_1 = X) &= P(X|X) \cdot P(X) + P(X|Y) \cdot P(Y) \\ &= 0.7 \cdot 0.4 + 0.2 \cdot 0.6 = 0.28 + 0.12 = 0.4. \end{aligned}$$

This means there is a 40% chance that the system's state at time t_1 is X. The probability of the state Y at time t_1 is calculated analogously:

$$\begin{aligned} P(s_1 = Y) &= P(Y|X) \cdot P(X) + P(Y|Y) \cdot P(Y) \\ &= 0.3 \cdot 0.4 + 0.8 \cdot 0.6 = 0.12 + 0.48 = 0.6. \end{aligned}$$

Note that $P(s_1 = X) + P(s_1 = Y) = 1$.

Matrix A Simplifies Notation So far, the calculations were expressed by conditional probabilities. These probabilities are contained in the transition matrix, A, and the probabilities of the initial states are contained in vector π. This simplifies the notation. More specifically, the probability that the state at time t_1 is $s_1 = X$ is expressed as follows:

$$P(s_1 = X) = a_{1,0}\pi_1 + a_{0,0}\pi_0$$

Here, π_0 is the probability that the initial state is X and $a_{0,1}$ is the probability of the transition from X to Y. Likewise, the probability of state Y at t_1 is expressed as follows:

$$P(s_1 = Y) = a_{0,1}\pi_0 + a_{1,1}\pi_1$$

Probability of a Concrete State at Time t_2 Let s_2 be the state at time t_2. State $s_2 = X$ can result from $s_1 = X$ with no subsequent change; or from $s_1 = Y$ with subsequent change from Y to x. Using, as before, the interplay between independent events and mutually exclusive events the probability $P(s_2 = X)$ is calculated as follows.

$$P(s_2 = X) = P(s_1 = X) \cdot a_{0,0} + P(s_1 = Y) \cdot a_{1,0}$$

Substituting for $P(s_1 = X)$ and $P(s_1 = Y)$ the results from the previous paragraph, we obtain the following:

$$P(s_2 = X) = [a_{1,0}\pi_1 + a_{0,0}\pi_0] \cdot a_{0,0} + [a_{0,1}\pi_0 + a_{1,1}\pi_1] \cdot a_{1,0}$$

By way of a simple exercise, the reader may want to express $P(s_2 = X)$.

Probability of a Concrete State at Any Time t_i The recursive nature of the calculations is now obvious. In the general case of a state at time t_i (for any i), the two probabilities are calculated from those known for the previous time, t_{i-1}:

$$P(s_i = X) = P(s_{i-1} = X) \cdot a_{0,0} + P(s_{i-1} = Y) \cdot a_{1,0}$$

$$P(s_i = Y) = P(s_{i-1} = X) \cdot a_{0,1} + P(s_{i-1} = Y) \cdot a_{1,1}$$

We will remember that the probabilities for each time are obtained from those for the previous time. If we are asked to calculate, say, $P(s_4 = X)$, we simply start with the probabilities of X and Y at time t_0, then use them to calculate the probabilities at time t_1, and so on until we reach t_4.

Once we get used to thinking in terms of this recursive pattern, we will find it easier to establish the probabilities in the *Hidden Markov Model, HMM.*

20.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What does the theory of probability understand under *mutually exclusive events*? What is understood under *independent event*?

- Derive the recursive formulas that calculate the probability of a concrete state at times t_0 and t_1.
- Explain the recursive nature of the mechanism to calculate the probability of a concrete state at any time, t_i.

20.3 HMM: Indirectly Observed States

Now that are comfortable with the probabilistic calculations involved in Markov processes, let us proceed to the full-fledged *Hidden Markov Models*, HMM. In this model, we do not have any certainty about the system's current state. The best we can do is to develop some judgment based on indirect observations.

Matrix of Observations For simplicity, suppose the system can be in only two different states, X and Y. We do not have direct access to them, but we make indirect observations that we know are to some degree dependent on these states. Suppose there are only two different observations, denoted as O_0 and O_1. The paradigm of the HMM rests on the assumption that we know the probability of any of the given observation in any state.

For instance, if the state is X, there may be a 20% chance of seeing O_0 and a 80% chance of seeing O_1; and if the state is Y, there may be a 90% chance of seeing O_0 and a 10% chance of seeing O_1. These probabilities are stored in the observation matrix, B:

$$B = \begin{bmatrix} 0.2 & 0.8 \\ 0.9 & 0.1 \end{bmatrix}$$

Here, states X and Y are represented by rows 0 and 1, respectively. Observations O_0 and O_1 are represented by columns 0, and 1, respectively. We will denote by $b_i(O_j)$ the probability of seeing O_j in the i-th state. For instance, $b_0(O_1) = 0.8$ (where the 0-th state is X).

Under the assumption that only the available observations can be made at any given state (that no other observations are possible), the values in each row of matrix B sum to 1.

Formal Definition of a HMM A *hidden Markov model* is defined by the initial-states vector, π, the transition matrix A, and the observations matrix, B. This is expressed by the following formula where λ is the HMM.

$$\lambda = (A, B, \pi) \tag{20.1}$$

The theory of HMM focuses on a few basic tasks. In the simplest of them, the values of the entries in A, B, and π are known, perhaps having been obtained by experimentation. At a more ambitious level, the entries are obtained by machine learning. The concrete procedure to estimate all these values is the ultimate goal of this chapter.

Table 20.1 Basic quantities of *hidden Markov models*. Note that all indices start from 0. For instance, if there are T time steps, the index of the last is $T-1$

1. Time steps, $t_0, t_1, \dots, t_{T-1}$. At time t_i, the system finds itself in state s_i.
2. The model's states, q_i with $i = 0, 1, \dots N-1$. At each time step, t_j, there is some i such that the system finds itself in state $s_j = q_i$.
3. Available observations, V_i, with $i = 0, 1, \dots M-1$.
4. A concrete sequence of observations, $\mathbf{O} = O_0, \dots, O_{T-1}$. For each observation, O_i, there is some j such that $O_i = V_j$.
5. State transition probabilities in matrix A.
6. Observation probabilities in matrix B.
7. Probabilities of initial states in vector π.

Quantities to Characterize a Given HMM Anybody's first encounter with HMM is discouraging because the number of the quantities employed in the analyses of a concrete HMM is so high. It is impossible not to be confused. Once the readers get used to all these symbols, however, the rest is easier than it seems.

For a quick reference, Table 20.1 summarizes the most important variables used in the sections to come. The table is no dogma. Whenever possible (when there is no danger of confusion), even simpler notation will be used.

Example of a HMM To get used to the meanings of some of these characteristics, consider the following example. Johnny plays in the local chess club. He plays either against strong players (S) or against weak players (W). These are the *states* of the HMM. For instance, the programmer can decide that $q_0 = S$ and $q_1 = W$. After a game with a strong player, Johnny chooses with certain probability a week player, or prefers another strong player; and likewise after a game with a weak player. These probabilities are stored in the transition matrix, A.

The opponent's strength affects the probability of each of the game's possible outcomes: V_0=win, V_1=loose, and V_3=draw. These outcomes are the HMM's *observations*. The probabilities of the concrete observations for strong and weak opponents are stored in the observation matrix, B.

Johnny plays one game a day. When he comes home, he tells his wife the result of the game, not the strength of the opponent. In other words, she gets the observation, O_i; for instance, on Monday she may learn that O_0=win. From a series of such observations, $\mathbf{O}$, made in the course of a week, and from the known π, A and B, she can calculate the probability that, say, the first three players were strong and the last three were weak.

Another Example of a HMM[2] Suppose that systematic measurements of average temperatures in a certain locality have been taken, and suppose these measurements

[2]This is borrowed from Mark Stamp's *Introduction to Machine Learning with Applications in Information Security*. CRC Press, Taylor & Francis Group, 2018.

indicate that some years are hot and others cold. `Hot` and `cold` represent the system's two states, q_0 and q_1. A hot year can be followed by another hot year, or by a cold year; likewise, a cold year can be followed by another cold year or by a hot year The probabilities of these changes are captured by the transition matrix A. The values in this matrix have been established by systematic observations recorded over the last century.

Research has established a probabilistic relation between temperatures and tree rings. The tree rings will be our observations: V_0=`small`, V_1=`medium`, and V_2=`large`. In hot years, they tend to be larger than in cold year, but such relations can only be probabilistic because the size of the tree rings may depend on other circumstances, not just average temperature. The observation matrix, B, has two rows, one for `hot` and one for `cold`, and it has three columns: for `small`, `medium`, and `large` tree rings. For instance, the entry in the first row and first column gives the probability of observing `small` tree rings in a `hot` year.

For distant past, the temperatures are not known, but tree rings are. If the system is modeled as an HMM, we can draw probabilistic conclusions about the temperatures in, say, the fifteenth century. True, we do not know the temperatures; but we do know the sizes of the tree rings, and all the requisite probabilities.

Why do We Need HMM? Once we know all the probabilities defining the given λ, we should be able to take the available sequences of observations, and use them to answer simple questions such as "What is the probability of a certain state at t_7?" or "What is the probability of a certain sequence of states, such as XYX?" or "Given a sequence of observations, what is the most likely series of the underlying states?" or "What are the probabilities of observation sequences?"

From the machine-learning perspective, the most interesting question will ask how to create the HMM-defining triplet, $\lambda = (A, B, \pi)$. More specifically, for the given training sequence of observations, what should the matrices A and B look like? This is addressed by an algorithm known as *Expectation Maximization*, EM. This algorithm is this chapter's ultimate goal.

Real-World Applications of HMM The field of application domains is remarkably rich. Thus in finance, one may want to be able to guess a sequence of events from observed changes in stock prices. In computer security, malware is known to consist of a series of states that are reflected in specific observed behavior of the affected computer. In bioinformatics, specific segments of a RNA are marked by series of created proteins, and thus the properties of the resulting tissues. And in computer video, one may want to reconstruct the original event based on an observed sequence.

Rich field of applications is provided by speech processing. Here is an example. In English, a word (e.g., "approximate") is pronounced differently depending on whether it is in the given context a verb or an adjective. The former is exemplified by "the approximate value of x is 5," and the latter is exemplified by "let us approximate the resulting value." For the English-speaking program to be able to decide which of the two pronunciations to choose, the whole text first needs to be *tagged*: each word has to be labeled as being `noun, adjective,` and so on. Of course, the tags do not follow each other at random; their time order follows certain patterns. These can then be captured in a HMM where the tags are *states* and the words are *observations*.

The reader has noticed one important thing: for the sake of simple explanation, textbook examples involve only two or three states and two or three observations. In realistic applications, these numbers are much higher.

20.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is *matrix of transitions*, and what is *matrix of observations*? How do we define a concrete HMM?
- Suggest an example of a HMM, different from the two mentioned in this section. What are its states and what are its observations?
- What typical questions will we want to answer using an HMM?

20.4 Useful Probabilities: α, β, and γ

This chapter began with some simple calculations related to plain Markov processes, just to make the reader accustomed to their recursive nature. This recursive nature is typical also of the field of HMM where, however, the math is more complicated.

Joint Probability of O_0 and q_i at $t = 0$ Within a given HMM, what is the probability that, at time t_0, the system is in the i-th state, q_i, and the observation O_0 is made?[3] More formally, what is the joint probability $P(O_0, s_0 = q_i)$?

Joint probability of X and Y is calculated as $P(X, Y) = P(Y|X)P(X)$. In our specific case, X is $s_0 = q_i$ and Y is O_0. The probability of the initial states is provided by the π-vector; for the i-state, it is π_i. Knowing that $P(O_0|s_0 = q_i)$ is available in the B-matrix as $b_i(O_0)$, we arrive at the following answer:

$$P(O_0, s_0 = q_i) = \alpha_0(i) = \pi_i \cdot b_i(O_0) \tag{20.2}$$

Note that the quantity is denoted here by $\alpha_0(i)$ where index 0 refers to the initial time step, t_0, and the argument i refers to the i-th state, q_i.

Probability of O_0 We have established the joint probability of O_0 and $s_0 = q_i$. The same observation could be made at *any* state. The system's different states are

[3]Following Table 20.1, we should say, "…probability that at t_0, the j-th observation is made, $O_0 = V_j$." The author believes he can simplify here the formalism without becoming ambiguous.

mutually exclusive events, which means that the overall probability of O_0 is the sum of the α-values for all individual states:

$$P(O_0) = \sum_{i=0}^{N-1} \alpha_0(i) \tag{20.3}$$

Joint Probability $P(O_0, O_1, s_1 = q_i)$ Let us consider the next time step, t_1. The observations at times $t = 0$ and $t = 1$ are O_0 and O_1, respectively. We want to know the joint probability of these observations *and* $s_1 = q_i$. In the spirit of the previous paragraph, the result will be denoted by $\alpha_1(i)$.

Again, the probability of O_1 being made in state q_i is available in the B-matrix as $b_i(O_1)$. State $s_1 = q_i$ can result from a set of mutually exclusive events: any initial state, $s_0 = q_j$, followed by the transition from q_j to q_i. The probability of this transition is provided in the A-matrix as $a_{j,i}$.

Recall that $\alpha_0(j)$ is the probability of O_0 *and* $s_0 = q_j$. Given that the initial observation was O_0, the probability of $s_1 = q_i$ is obtained by multiplying the values of $\alpha_0(j)$ (for each $s_0 = q_j$) by the probability, $a_{j,i}$, of the transition from q_j to q_i.

$$P(O_0, s_1 = q_i) = \sum_{j=0}^{N-1} \alpha_0(j)\ a_{j,i} \tag{20.4}$$

The B-matrix informs us that for any state q_i, the probability of O_1 is $b_i(O_1)$. Combined with the last equation, we can establish the joint probability of O_0, O_1 and $s_1 = q_i$ as follows:

$$\alpha_1(i) = [\sum_{j=0}^{N-1} \alpha_0(j)\ a_{j,i}] \cdot b_i(O_1) \tag{20.5}$$

Calculating *Alpha*: Joint Probability $P(O_0, O_1, \ldots, O_t, s_t = q_i)$ The reasoning from the previous paragraphs is easily generalized to all the subsequent time steps. It can be easily proved that the value of $\alpha_t(i)$ can be calculated from $\alpha_{t-1}(i)$, which is the value calculated for the previous time step, $t - 1$:

$$\alpha_t(i) = [\sum_{j=0}^{N-1} \alpha_{t-1}(j)\ a_{j,i}] \cdot b_i(O_t) \tag{20.6}$$

Probability of a Given Series of Observations We are ready for one of the major results. To wit, for the given HMM defined by some λ, we are able to establish the probability of any sequence of observations, $\mathbf{O} = (O_0, O_1, \ldots, O_t)$. For this, we only need to sum, at the final time step $T-1$, the alphas for all the individual states (they are mutually exclusive events):

$$P(\mathbf{O}|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i) \tag{20.7}$$

Defining *Beta* Whereas α involves the sequence of observations from the initial state to some $t = T$, quantity β deals with the sequence of observations from some time step t to the end of the sequence. Let to define this more formally.

Suppose that, at time step t, the system's state is $s_t = q_i$. What is the probability that this state will be followed by the sequence of observations, $O_{t+1}, O_{t+2}, \ldots, O_{T-1}$? We want to know the value of the following conditional probability:

$$\beta_t(i) = P(O_{t+1}, O_{t+2}, \ldots, O_{T-1}|s_t = q_i) \tag{20.8}$$

Calculating *Beta* Suppose that, at time t, the system is in state $s_t = q_i$. In the next step, this state is converted to any other state, q_j, with probability $\alpha_{i,j}$ known from the A-matrix. In any of these new states, the probability of observation O_{t+1} being made in the new state is $b_j(O_{t+1})$. These new states being mutually exclusive events, the probability of the given observation O_{t+1} is obtained by the following sum:

$$P(O_{t+1}|s_t = q_i) = \Sigma_{j=0}^{N-1} \; a_{i,j} b_j(O_{t+1}) \tag{20.9}$$

Multiplying this by $\beta_{t+1}(j)$, which is the probability of all the remaining observations, $O_{t+2}, \ldots, O_{T-1}$ for the next state, $s_{t+1} = q_j$, we obtain the following recursive formula for the calculation of β:

$$\beta_t(i) = \Sigma_{j=0}^{N-1} \; a_{i,j} b_j(O_{t+1}) \beta_{t+1}(j) \tag{20.10}$$

Defining and Calculating *Gamma* Again, suppose that in a HMM specified as $\lambda = (A, B, \pi)$, the following series of observation was made: $\mathbf{O} = (O_0, O_1, \ldots, O_t)$. What is the probability that, at time t, the system's state is $s_t = q_i$? The answer follows from the previous definitions of $\alpha_t(i)$ and $\beta_t(i)$. The former gives the probability of the given observations until time t, the latter gives the probability of the remaining observations after time t. These being independent of each other, the resulting probability is the product of the two probabilities.

As we want to establish the probability of $s_t = q_i$ only within the given series of observations, $\mathbf{O}$, we must not forget to divide the result by the probability that this series of observations is made, under the given HMM. This probability is written as $P(\mathbf{O}|\lambda)$:

$$\gamma_i(i) = \frac{\alpha_t(i) \cdot \beta_t(i)}{P(\mathbf{O}|\lambda)} \tag{20.11}$$

The attentive reader will recall that Eq. (20.7) established that the probability in the denominator is obtained as the sum of all the alphas at the final time step, $T-1$:

$$P(\mathbf{O}|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$$

20.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- In plain English, explain the meaning of the probabilities established here by the calculations of α and β.
- Derive the recursive formulas for the calculations of α and β.
- What is the meaning of γ? Write down the formula calculating its value and explain why it looks like this.

20.5 First Problem and Second Problem of HMM

The probabilities denoted by α and β provide the foundations that help us solve what is known as the *first problem of HMM* and the *second problem of HMM*. Let us explain the concrete algorithms.

First Problem of HMM Given a hidden Markov model $\lambda = (A, B, \pi)$, what is the probability that the series of observations $\mathbf{O} = (O_0, \ldots, O_{T-1})$ will be made?

Mathematically speaking, the solution is provided by Eq. (20.7). However, the value of $\alpha_{T-1}(i)$ is not immediately known, and can only be established recursively. This is the task for the algorithm whose pseudo-code is in Table 20.2. The procedure is known as the *forward pass*, a term that reminds us of the way the α-probabilities are propagated from the initial time step, t_0, to the final time step, t_{T-1}.

At the beginning, we use Eq. (20.2) to establish α-values at the initial time step for all states, q_i. From these, the α-values for time $t = 1$ can be obtained by Eq. (20.6), and so on until the final time step, $T-1$, is reached. Once all values of $\alpha_{T-1}(i)$, are known, the probability of the given series of observations, $\mathbf{O}$, is obtained by Eq. (20.7).

Backward Algorithm* to Find *Beta Before proceeding to the second HMM-problem, we need an algorithm to calculate all β-values. The pseudo-code is in Table 20.3.

Table 20.2 Solution to the first problem of HMM by the *forward algorithm*

Problem 1.
For a given $\lambda = (A, B, \pi)$, what is the probability of observing $\mathbf{O} = (O_0, \dots, O_{T-1})$?
Solution.

1. For all states q_i, calculate the $\alpha_0(i)$ at the initial time step:

$$\alpha_0(i) = \pi_i \cdot b_i(O_0)$$

2. For all the remaining time steps, $t = 1, \dots, T-1$, and for all states q_i, calculate $\alpha_t(i)$ recursively by the following formula:

$$\alpha_t(i) = [\sum_{j=0}^{N-1} \alpha_{t-1}(j)\ a_{j,i}] \cdot b_i(O_t)$$

3. Calculate $P(\mathbf{O}|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i)$

Table 20.3 *Backward algorithm* to calculate β-values

Solution.
$\beta_t(i)$ is the probability of that the sequence of observations, $\mathbf{O} = (O_{t+1}, \dots, O_{T-1})$, will be made if, at time t, the system is in state $s_t = q_i$.

1. $\beta_{T-1}(i) = 1;$
2. $\beta_{T-2}(i) = \Sigma_{j=0}^{N-1}\ \ a_{i,j} b_j(O_{T-1})\ \beta_{T-1}(j)$

$\vdots$

3. $\beta_t(i) = \Sigma_{j=0}^{N-1}\ \ a_{i,j} b_j(O_{t+1})\ \beta_{t+1}(j)$

We know that, for any given state q_i, the value of $\beta_t(i)$, can only be established recursively because, for each value, we first need to know the value in the next time step, $\beta_{t+1}(i)$. This was established in Eq. (20.10).

At the same time, it is obvious that, in the last time step, $T-1$, this probability has to be 100%. The algorithm to find all β-values therefore starts by initialing $\beta_{T-1}(i) = 1$ for all states, q_i. These values are then backpropagated to time $T-2$, then $T-3$, all the way down to $\beta_t(i)$ at time t.

Second Problem of HMM Given a hidden Markov model $\lambda = (A, B, \pi)$, and a series of observations, $\mathbf{O} = (O_0, \dots, O_{T-1})$ what is the probability that the system has gone through the series of states, $q_0, \dots, q_{T-1}$?

One of the useful quantities introduced in the previous section is $\gamma_i(i)$, the probability that, at time t, the system is in state $s_t = q_i$ (for the given HMM and the given series of observations). Equation 20.11 shows how to calculate this probability from the known α-values and β-values. In view of this, the solution of HMM's second problem does not present any difficulties.

Table 20.4 Solution to the second problem of HMM

Problem 2.
For a given $\lambda = (A, B, \pi)$, and a series of observations, $\mathbf{O} = (O_0, \ldots, O_{T-1})$, what is the probability of a given series of states, $q_0, \ldots, q_{T-1}$?
Solution.

1. For each s_t in the series, calculate the values of $\alpha_t(i)$ and $\beta_t(i)$.
2. Calculate the probability of the given observation, $P(\mathbf{O}|\lambda)$.
3. Calculate, $\gamma_t(i) = P(s_t = q_i|\mathbf{O}, \lambda)$: the probability that the system is at time t in state q_i:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(\mathbf{O}|\lambda)}$$

4. The probability of the given sequence of states is the product of the γ-values of the individual states in this sequence.

As indicated in the pseudo-code in Table 20.4, it is enough to calculate α-values and β-values for each state in the given sequence and then use them to calculate the corresponding γ-value for each state. Given that the states in different time steps are independent events, the probability of the entire sequence is the product of the probabilities of the individual states.

Most Likely State at Time *t* The knowledge of γ-values can be used in the search for the most likely sequence of states underlying a given series of observations. To find the answer, we simply calculate for each time step t, the most likely state:

$$X_t^{(best)} = \max_i \gamma_t(i) \tag{20.12}$$

The most likely sequence of states is the sequence of X_t^{best}'s for all time steps t. The term, "most likely" in this context means the minimum number of differences between the true states at times t, and the states established as $X_t^{(best)}$ by the previous equation.

20.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the definition of the *first problem of HMM*? Summarize the *forward algorithm* capable of solving it.
- Summarize the *backward algorithm* that establishes the β-values for a given $s_t = q_i$ and for a given series of observations.
- What is the definition of the *second problem of HMM*? Summarize the algorithm capable of solving it.
- How would you find the most likely sequence of states underlying the given series of observations?

20.6 Third Problem of HMM

Up till now, this chapter always assumed that the probabilities in A, B and π are known. This, however, is not necessarily the case. Fortunately, they can be estimated. The concepts developed in the previous sections will help us address the HMM's ultimate challenge: how to use observational data in creating the model.

Third Problem of HMM For a given series of observations, $\mathbf{O} = (O_0, \ldots, O_{T-1})$, find the model, $\lambda = (A, B, \pi)$.

The reader can see the difference between this problem and the previous two. Whereas the previous two focused on practical applications of a known HMM, this one seeks to use available observations to *create* the underlying HMM by estimating the values in the A-matrix, B-matrix, and π-vector. Strictly speaking, only the third problem (and not the first two) is a machine-learning task.

Defining Di-gamma The presentation of the solution will be easier if we introduce yet another "Greek probability": di-gamma, which is something similar to the γ-value from the previous section, but with two arguments instead of just one.

For a given $\lambda = (A, B, \pi)$ and a series of observations, $\mathbf{O} = (O_0, \ldots, O_{T-1})$, we define $\gamma(i, j)$ as the probability that a system will at time step t be in state $s_t = q_i$, and at time step $t + 1$ in state $s_{t+1} = q_j$. The reader by now experienced enough to establish the formula to calculate this quantity:

$$\gamma_t(i, j) = \frac{a_t(i)\ a_{i,j}\ b_j(O_{t+1})\ \beta_t(j)}{P(\mathbf{O}|\lambda)} \tag{20.13}$$

The relation between $\gamma_t(i)$ and $\gamma_t(i, j)$ is obvious:

$$\gamma_t(i) = \Sigma_{j=0}^{N-1}\ \gamma_t(i, j) \tag{20.14}$$

Expectation Maximization To begin with, $\lambda = (A, B, \pi)$ is initialized—in the simplest case, to random values. After this, the solution to HMM's third problem is obtained in the course of a series of alternations between two steps: *expectation* and *maximization*.

In the first step, *expectation*, existing values of A and B are used to calculate the probabilities $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i)$, and $\gamma_t(i, j)$ that were introduced in the previous sections. In the second, these values are used to predict the most likely observations; these being different from the real observations, some corrections of λ are necessary. These corrections are made by the second step, *maximization* (Table 20.5).

Re-calculating A-Matrix Let us remind ourselves of what the A-matrix represents: the entry $a_{i,j}$ gives the probability of transition from state q_i to state q_j. Based on the current version of the HMM, and based on the given series of observations,

Table 20.5 *Expectation maximization* for the solution of the third problem of HMM

Problem 3.

For a given series of observations, **O**, find the model, $\lambda = (A, B, \pi)$.

Solution.

1. Initialize $\lambda = (A, B, \pi)$. In the simplest implementation, random values are used.
2. Compute the probabilities $\alpha_t(i)$, $\beta_t(i)$, $\gamma_t(i)$, and $\gamma_t(i, j)$ for all time steps t, and for all states q_i and q_j.
3. Re-estimate $\lambda = (A, B, \pi)$ based on the observations.
4. If $P(\mathbf{O}|\lambda)$ has increased, return to step-2. Otherwise, stop.

Note. Step-2 represents *expectation* and step-3 represents *maximization*.

this is recalculated as the number of observed transitions from state q_i to state q_j, divided by the number of times the system was in state q_i:

$$a_{i,j} = \frac{\Sigma_{t=0}^{T-2} \gamma_t(i, j)}{\Sigma_{t=0}^{T-2} \gamma_t(i)} \tag{20.15}$$

Note that the summations have to stop at $t = T - 2$ which is the second-to-last time instant because there is no transition from the last state, s_{T-1}.

Re-calculating B-Matrix Let us remind ourselves of what the B-matrix represents: the entry $b_i(O_j)$ is the probability of seeing O_j in state i. Based on the current version of the HMM, and based on the given series of observations, these probabilities are estimated as follows:

$$b_j(k) = \frac{\Sigma_{O_t=k;}\, \gamma_t(j)}{\Sigma_{t=0}^{T-2} \gamma_t(j)} \tag{20.16}$$

Note that the denominator sums all γ's, whereas the numerator sums the γ's only for times in which observation O_k was made. Put another way, the fraction divides the number of "correct" observations by the number of all observations.

20.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is known as the *the third problem of HMM*? In what way is it different from the previous two?
- Explain the principle of the *Expectation Maximization* approach as employed in this context. Which of its steps is *expectation* and which is *maximization*?
- What is *di-gamma*? Write down the formula that calculates its value.
- How exactly are the values in the A-matrix and B-matrix updated? Write down the formulas.

20.7 Summary and Historical Remarks

- *Markov process* is a system that, in discrete time intervals, proceeds from one state to another, but each state depends only on the previous state, and on older history.
- *Hidden Markov Models*, HMM, generalize Markov processes in the sense that we now do not have direct access to the systems' states, but only to observations from which the probabilities of the individual states can be inferred.
- HMM is defined as the triplet $\lambda = (A, B, \pi)$ where A is the matrix of transitions, B is the matrix of observations, and vector π gives the probabilities of the initial states.
- The three fundamental problems of HMM include (1) the probabilities of observation sequences, (2) the probabilities of state sequences, and (3) the question how to establish the entries in the A-matrix and B-matrix.
- To help answer the fundamental problems, mathematicians have identified some critical probabilities, known as α. β, γ, and di-γ.
- For the solution of HMM's first problem, the *forward algorithm* finds the α-values or all states at the final time step. The sum of these α-values then gives the probability of the given observation sequence.
- For the solution of HMM's second problem, the *backward algorithm* finds the β-values for all states and times. Together with α-values, these are then used in a formula that calculates $\gamma_t(i)$ which is the probability of q_i at time step t. The probability of a given sequence of states is the product of these $\gamma_t(i)$'s found in the individual time steps.
- The third problem of HMM is addressed by the *expectation maximization* algorithm. Based on observational data, *expectation* calculates all values of α. β, γ, and di-γ. Once this is done, *maximization* seeks to improve the values in the A-matrix and B-matrix.

Historical Remarks The paradigm of *Hidden Markov Models* predates modern machine learning. The basic theoretical model was first studied by Baum and Petrie (1966), but the major impetus came with its application to speech recognition by Baker (1975). The author has learned a lot from the book by Stamp (2018).

20.8 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

20.8.1 Exercises

1. Suppose that a Markov process is defined by the following probabilities of state transitions and initial states:

$$A = \begin{bmatrix} 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix} \qquad \pi = [0.4, \ \ 0.6]$$

   The state corresponding to index 0 is **x** and the state corresponding to index 1 is **y**. What is the probability that the system's state at $t = 1$ will be **y**?
2. Add to the A-matrix and π-vector from the previous exercise also the following B-matrix:

$$B = \begin{bmatrix} 0.2 & 0.8 \\ 0.9 & 0.1 \end{bmatrix}$$

   For instance, the probability of seeing O_1 in state q_0 is 80%.

   What is the probability that the first two observations in the series are O_1 and again O_1, and that the state at $t = 1$ is $s_1 = q_0$?

20.8.2 Give It Some Thought

1. Suggest an example (different from those mentioned in this chapter) of a system that can be analyzed using the formalism of HMM. What are its states and observations? What sequences of states and observations can the system learn from?
2. In the *maximization* step of the solution of HMM's third problem, the values in the A and B matrices are updated using Eqs. (20.15) and (20.16). Give a verbal explanation of why these formulas indeed tend to improve the contents of these two matrices. Of course, your explanation should go deeper than the text in Sect. 20.6.
3. Suggest a stopping criterion for the algorithm that solves the third problem of HMM in Sect. 20.6 (not just the pre-specified number of iterations).

20.8.3 Computer Assignments

1. Write a Markov-process program that, using the A-matrix and the π-vector, will calculate the probability of any state at any time step.
2. Consider a HMM described by some $\lambda = (A, B, \pi)$. Write a program that for a given series of observations calculates all α-values and β-values.

3. Write a program that uses the α-values and β-values obtained in the previous step to solve the first and second problems of HMM.
4. Write a program implementing the solution to the third problem of HMM. Pay attention to the program's input parameters: series of observations, initial values of $\lambda = (A, B, \pi)$, and perhaps the number of iterations to serve as the stopping criterion.

Chapter 21
Genetic Algorithm

Machine learning, in all its methods and applications, essentially searches for a best solution to a given task. The goal can be a classifier, a technique to optimize an automated agent's behavior, or even some knowledge gleaned from unlabeled data. In the past, many of these problems were addressed by search techniques borrowed from the field of Artificial Intelligence.

This chapter presents a powerful alternative: the *genetic algorithm*, inspired by the principles of Darwinian evolution. While the principle is easy to understand, the paradigm is flexible enough to be applied to a broad range of problems. Once the baseline version has been explained, and its behavior understood, the chapter shows how to employ it in a typical machine-learning application.

21.1 Baseline Genetic Algorithm

Let us first briefly describe the general principle, relegating the details of its implementation to the next section.

Underlying Philosophy This section assumes that an engineering problem can be represented by a *chromosome*, typically a string of bits that are sometimes referred to as "genes." The genetic algorithm operates with a *population* of chromosomes, each describing one individual which can be a classifier or an agent capable of responding adequately to changes in its environment or any other machine-learning application.

Fitness function assigns to each individual a value that quantifies the chromosome's performance. Different fields are accustomed to different terminology. Instead of "fitness function," some literature prefers the terms "survival function" or "evaluation function."

Genetic Algorithm's Endless Loop The principle is shown in Fig. 21.1. At each moment, there is a population of individuals, each with a survival value calculated

M. Kubat, *An Introduction to Machine Learning*,
https://doi.org/10.1007/978-3-030-81935-4_21

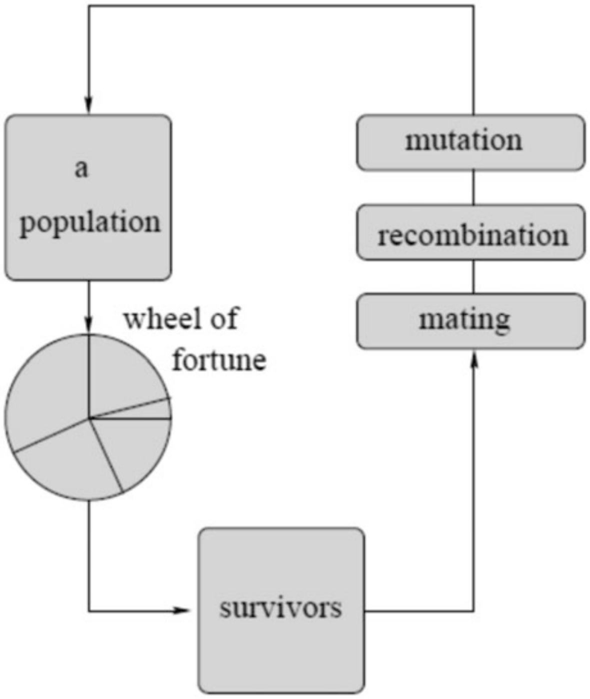


Fig. 21.1 Genetic algorithm's endless loop. Each individual has a certain chance of survival. Survivors find mating partners and generate new individuals by recombining their genetic information. Chromosomes can be corrupted by mutation

by the fitness function; this value determines the size of the segment assigned to the individual in a "wheel of fortune." Fate throws darts at the wheel, each throw choosing for survival one individual. It is important to appreciate the probabilistic nature of the process. While the chances of an individual with a large segment are high, the game is non-deterministic. Just like in the real world, a specimen with excellent genes may perish in a silly accident, while a weakling can make it just by good luck. It is only in the long run, and in a large population, that the laws of probability succeed in favoring the genes that promise high fitness.

The surviving specimens then choose "mating partners." In the process of mating, the chromosomes of the participating individuals are *recombined*, and this gives rise to a pair of new chromosomes. These new chromosomes may subsequently be subjected to *mutation*, which adds noise to the strings of genes.

The whole principle is summarized by the pseudo-code in Table 21.1.

How the Endless Loop Works Once a new population has been created, the process enters a new cycle in which the individuals are subjected to the same wheel of fortune, followed by mating, recombination, and mutation. The story goes on and on until stopped by an appropriate termination criterion.

Again, note the probabilistic nature of process. A low-quality chromosome may survive the wheel of fortune by a fluke; but if a child's fitness remains low, the genes will perish in subsequent generations. Alternatively, however, some of an

Table 21.1 Pseudo-code of the genetic algorithm

```
initial state: a population of chromosomes (“individuals”)
1. The fitness of each individual is evaluated. The fitness values then decide (probabilistically)
   about each individual’s survival.
2. Survivors find mating partners.
3. New individuals (“children”) are created by the recombination of the chromosomes of the
   mating partners.
4. Chromosomes are corrupted by random mutation.
5. Unless a termination criterion is satisfied, the algorithm returns to step 1.
```

unpromising individual’s genes may prove to be useful when embedded in the context of different chromosomes resulting from recombination. By giving them an occasional second chance, the process enjoys remarkable flexibility.

21.1.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the principle of the genetic algorithm. How are individuals described? What is meant by the term, “survival chance”?
- Summarize the basic loop of the genetic algorithm.
- What is the advantage of the probabilistic nature of the survival as compared to its hypothetical deterministic implementation?

21.2 Implementing the Individual Functions

How to implement the genetic algorithm in a computer program that consists of the survival game, mating process, chromosome recombination, and mutation? Let us discuss here only some simple solutions, relegating more advanced techniques to later sections.

To begin, we will consider chromosomes in the form of binary strings, such as `[1 1 0 1 1 0 0 1]`, where each bit represents a property that is either present, in which case the value is `1`, or absent, in which case it is `0`. Thus in a simplified version of the “pies” problem, the first bit may indicate whether or not `crust` is `thick`, the second bit may indicate whether or not `filling` is `black`, and so on.

Initial Population The most common way of creating the initial population employs a random-number generator. Sometimes, the engineer can rely on some knowledge that may help her create initial chromosomes known to be more representative than randomly generated strings. In the “pies” domain, this role can be played by the descriptions of the positive examples. However, what we need is an initial population that is sufficiently large and has sufficient diversity.

Fitness Function Genetic algorithms assume that there is a way to estimate for each specimen its survival chances. In some applications, these chances can be obtained by experimentation such as in game playing where the the individuals can "fight it out." In other domains, survival fitness is calculated by a user-specified function whose output depends on the chromosome's contents. For instance, if the chromosome represents a classifier, the fitness function may return the percentage of the training examples that the classifiers labels correctly.

Implementing the Wheel of Fortune An individual's survival is determined probabilistically. Here is how to implement the "wheel of fortune." Let F_i denote the i-th individual's fitness so that $F = \Sigma_i F_i$ is the sum of all individual's fitness values. These are then arranged along the interval $(0, F]$. A random-number generator is asked to return some $r \in (0, F]$: the sequential number of the sub-interval "hit" by r then points to the winner.

The method is illustrated by Fig. 21.2 for a population of four specimens and a random number landing in the third interval: individual 3 is thus selected. If we want 20 specimens to survive, we generate 20 random numbers whose locations in $(0, F]$ decide which individuals to copy in the "pool of survivors."

Whereas specimens with small fitness are thus likely to get eliminated, those with higher fitness can appear in the pool of survivors more than once. In biology, an individual can survive only once, but the pragmatic world of computer programmers easily accepts the possibility that the same individual "survives" twice, three times—or many times.

Mating Operator The survival game is followed by mating. In nature, an animal chooses its partner by such criteria as strength, speed, or sharp teeth. Something similar is accomplished in a computer implementation with the help of the fitness function. There is a difference, though: the notion of gender is usually ignored—any chromosome can mate with any other chromosome.

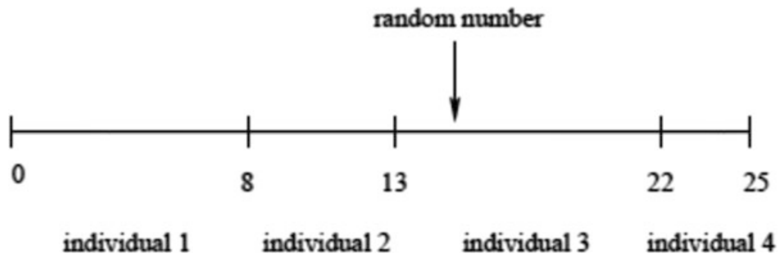


Fig. 21.2 The axis represents a population of four individuals whose fitness values are 8, 5, 9, and 3, respectively. Since the randomly generated number, 15, falls into the third sub-interval, the third individual is selected

An almost trivial mating strategy will pair the individuals arbitrarily by generating random pairs of integers from interval $[1, N_s]$ where N_s is the size of the population. This fails to do justice to the requirement that specimens with high fitness should perhaps be deemed more attractive. A simple way to reflect this in a computer program is to order the individuals in the descending order of their fitness, and then to pair the neighbors in this list.

Yet another strategy proceeds probabilistically. It takes the highest-ranking individual, then chooses its partner using the mechanism employed in the survival game—see Fig. 21.2. The same is done for the second highest-ranking individual, then for the third, and so on, until the new population has reached the required size. Higher-valued individuals are thus likely (though not guaranteed) to mate with other high-valued individuals. Sometimes, the partner will have a low quality (due to the probabilistic selection), but this gives rise to diversity that allows the system to preserve valuable chromosome chunks even if they have the bad luck of currently sitting in low-quality specimens.

Chromosome Recombination: One-Point Crossover The simplest way to implement chromosome recombination is by the *one-point crossover* operator that swaps parts of the parent chromosomes. Suppose that each chromosome consists of n bits and that a random-number generator has returned integer $i \in [1, n]$. Then, the last i bits in the first chromosome (its i-bit tail) are replaced with the last i bits in the second chromosome and vice versa. A concrete implementation can permit the situation where $i = n$, in which case the two children are just replications of their parents. In the example below, the random integer is $i = 4$, which means that 4-bit tails are exchanged (the crossover point is indicated by a space).

$$\begin{array}{c} 1101\ 1001 \\ 0010\ 0111 \end{array} \Rightarrow \begin{array}{c} 1101\ 0111 \\ 0010\ 1001 \end{array}$$

The operator is meant to ensure that the children are similar to their parents. This is the case especially when the exchanged tails are short.

In some applications, the recombination operator is applied only to a certain percentage of individuals. For instance, if 50 pairs have been selected for mating, and if the probability of recombination has been set by the user as 80%, then only 40 pairs will be subject to recombination, and the remaining 10 will just be copied into the next generation.

Mutation Operator The task for mutation is to corrupt the inherited genetic information. Practically speaking, this is done by “flip-flopping” a small percentage of the bits in the sense that `0` is changed to `1` and the other way round. The concrete frequency of these mutations is set by the user. Suppose that this parameter requires that $p = 0.001$ of the bits should on average be mutated. The corresponding program module will then for each bit generate a random integer from the interval [1, 1000]. If this integer equals 1, then the bit’s value is changed, otherwise it is left alone.

Consequence of Mutation What frequency of mutations we need? At one extreme, very rare mutations will hardly have any effect. At the other, high mutation frequency may disrupt the genetic search by damaging too many chromosomes. If the frequency is 100%, then mutation only creates a mirror image of the current population. Mutation probability of 50% essentially causes the genetic algorithm to degenerate to a random-number generator.

Mutation serves a different purpose from that of recombination. In the one-point crossover, no new information is created, only existing sub-strings are swapped. Mutation introduces some new twist that may result in something previously absent in the population.

Long-Living and Immortal Individuals One shortcoming of the above procedure is that very good individuals may be replaced by lower-valued children, which means that a good solution may get lost, even if only temporarily. To prevent this from happening, we sometimes instruct to program to copy the best specimens into the pool of survivors alongside their children. For instance, the program may directly insert in the new generation 20% best individuals, and then create the remaining 80% by applying the recombination and mutation operators to the remaining individuals. One may also consider the possibility of ignoring, say, the bottom 5%. In this way, not only will the best specimens live longer (they may even become "immortal"), but the program may be forced to get rid of very weak specimens that have survived by mere chance.

21.2.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the main task of the survival game and how would you implement it in a computer program?
- Describe a simple mechanism to implement the selection of the mating partners. Describe the recombination operator, and the mutation operator.

21.3 Why It Works

Let us now offer an intuitive explanation of why the genetic algorithm so often succeeds in finding good solutions to engineering problems.

Function Maximization The goal of the simple problem in Table 21.2 is to find the value of x for which the function $f(x) = x^2 - x$ is maximized. Each chromosome in the second column of the upper table is interpreted as a binary-encoded integer whose decimal value is given in the third column. The fourth

Table 21.2 One generation in the genetic algorithm

Suppose we want to find the maximum of $f(x) = x^2 - x$. Let x be an integer represented by a binary string. The initial population consists of the four strings below. For each string, the table gives its integer value, x, the value of function $f(x)$, the survival chances (proportional to $f(x)$), and the number of times each exemplar was selected (by the survival game) for the next generation.

No.	Initial population	x	$x^2 - x$	Survival chance	Actual count
1	0 1 1 00	12	132	0.14	1
2	1 1 0 0 1	25	600	0.50	2
3	0 1 0 0 0	8	56	0.05	0
4	1 0 0 1 1	19	342	0.31	1
Average			<u>282</u>		
Maximum			<u>600</u>		

Suppose the neighboring specimens mated, exchanging 1-bit tails and 3-bit tails, respectively, as dictated by the randomly generated tail lengths (the crossover sites are indicated by spaces). No mutation was used here. The last two columns give the values of x and $f(x)$ for the new generation.

After reproduction	Mate with	Tail length	New population	x	$x^2 - x$
0 1 1 0 0	2	1	0 1 1 0 1	13	156
1 1 0 0 1	1	1	1 1 0 0 0	24	552
1 1 0 0 1	4	3	1 1 0 1 1	27	702
1 0 0 1 1	3	3	1 0 0 0 1	17	289
Average					<u>425</u>
Maximum					<u>702</u>

The reader can see that the value of the best specimen and the average value in the entire population have increased.

column gives the corresponding $f(x)$ whose relative value, shown in the fifth column, then determines for each individual its survival chances. For example, the first specimen has $f(x) = 12^2 - 12 = 132$ and the relative chances of survival (in this population) are 14% because $132/(132 + 600 + 56 + 342) = 0.14$. The rightmost column tells us how many times each individual has been selected (by the wheel-of-fortune survival game) for inclusion in the next generation.

In the next step, the survivors choose their mating partners. Let us simply pair the neighboring specimens: the first with the second, and the third with the fourth. Then, the random selection of the crossover point decides that 1-bit tails be exchanged in the first pair and 3-bit tails in the second. No mutation is applied. The result is shown in the bottom table where the last three columns show, respectively, the new binary strings, their decimal values, and the values of $f(x)$. Note that both the average and the maximum value of the fitness function have increased.

Do Children Always Outperform Their Parents? Let us ask what caused this improvement. An intuitive answer is provided by Fig. 21.3 that shows the location

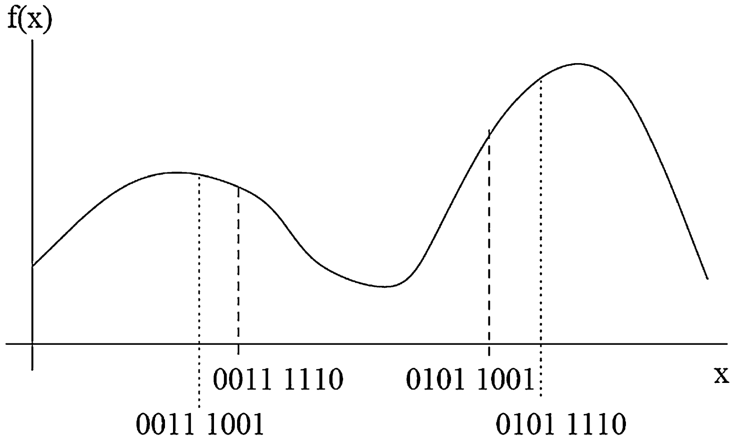


Fig. 21.3 After exchanging 4-bit tails, two parent chromosomes (upper strings) give rise to two children (lower strings). There is a chance that at least one child will "outperform" both parents

of two parents and the values of the survival function, $f(x)$, for each of them (dashed vertical lines). When the two chromosomes swap their 4-bit tails, two children are created, each relatively close to one of the parents. The fact that the children have higher $f(x)$ than the parents begs the question: is this always the case?

Far from that. All depends on the length of the exchanged tails and on the shape of the fitness function. Imagine that in the next generation the same two children get paired with each other and that the randomly generated crossover point is at the same location. Then, these children's children will be identical to the two original strings (their "grandparents"). This means that the survival chances decreased back to the original values. Sometimes, both children outperform their parents; in other cases, they are weaker than their parents; and quite often, we get a mixed bag. What matters is that in a sufficiently large population, the better specimens are more likely to survive; the selection process favors individuals with higher fitness, $f(x)$. Unfit specimens will occasionally make it, but they lose in the long run.

Are Children Similar to Parents? If the exchanged tails are short, the children's decimal values are close to those of their parent chromosomes; long tails cause the children to be much less similar to parents. As for mutation, its impact on the distance between the child and its parent depends on which bit is mutated. If it is the leftmost bit, the mutation causes a big jump along the horizontal axis. If it is the rightmost bit, the jump is short. Either way, mutation complements recombination. Whereas the latter tends to explore the space in the vicinity of parent chromosomes, the former may look elsewhere.

Shape of the Fitness Function Figure 21.4 illustrates two pitfalls related to the fitness function. The function on the left is almost flat. The fact that different individuals have virtually the same survival chances then defeats the purpose of the

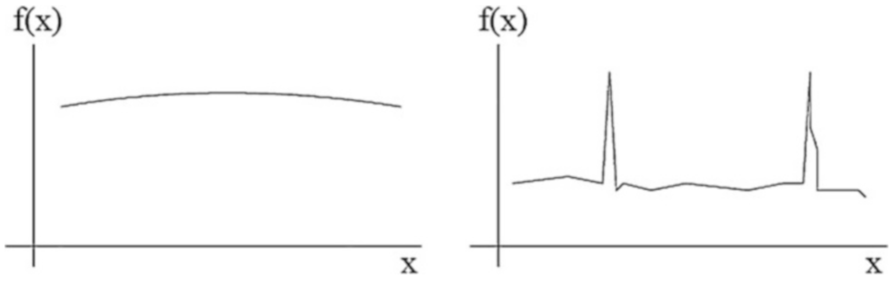


Fig. 21.4 Examples of two fitness functions that are poor guides for the genetic search. The one on the left is too flat, and the one on the right contains isolated narrow peaks

survival game. When the survivors are selected from a near-uniform distribution, the qualities of the winners will not give them perceptible advantage. This drawback can be mitigated by making $f(x)$ less flat. There are many ways to do so, one possibility being to replace $f(x)$ with, say, $f(x) = f^2(x)$.

The function in the right-hand part of Fig. 21.4 is marked by isolated narrow peaks. If the parent lies just at a hill's foot, the child may find itself on the opposite side of the peak which, in this case, will go unnoticed. We see that we need some gradual ascent toward the function's maximum. This second pitfall is more difficult to prevent than the previous one.

21.3.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain how the location of the crossover point determines how much the children differ from their parents.
- Explain how the mutual interplay between recombination and mutation may affect the survival chances. Show, also, how they depend on the concrete shape of the survival function and on the location of the parents.

21.4 Premature Degeneration

Suppose the genetic algorithm has reached a value that does not seem to improve over a series of generations. Does this mean that the ultimate solution has been found? Not really. The plateau may be explained by other circumstances.

Premature Degeneration Simple implementations stop the algorithm after a predefined number of generations. A more sophisticated version will keep track of the highest fitness achieved so far, and then terminate the search if this value no longer improves.

There is a catch, though. The fact that the fitness value has reached a plateau may not guarantee that a solution has been found. Rather, the plateau may be indicative of the so-called *premature degeneration*. Suppose that the search from Table 21.2 resulted in the following population:

```
0 1 0 0 0
0 1 0 0 1
0 1 0 0 0
0 1 0 0 0
```

What are the chances of further improvement? Recombination will not get us anywhere. If the (identical) last two chromosomes mate, their children will only copy their parents. If the first two are paired, then 1-point crossover will only swap the rightmost bit, an operation that does not create a new chromosome, either. The only chance of progress is offered by mutation which, if it affects appropriate bits, can reignite the process. For instance, this may happen after the mutation of the third bit in the first chromosome and the fourth bit (from the left) of the last chromosome. Unfortunately, mutations are rare, and to wait for the useful ones to happen may prove impractical.

Preventing Premature Degeneration Premature degeneration has a lot to do with the population's *diversity*. The worst situation is the one in which all chromosomes have exactly the same bit strings, something the engineer wants to avoid. Any computer implementation will therefore benefit from a module that monitors the population and takes action whenever its diversity drops below a certain level. A simple way to do so is to calculate the average similarity between pairs of chromosomes, perhaps by counting the bits with the same value in both strings. For instance, the similarity between `[0 0 1 0 0]` and `0 1 1 0 0]` is 4 (four bits are equal) and the similarity between `[0 1 0 1 0]` and `[1 0 1 0 1]` is 0.

Reduced diversity is not yet a reason for alarm. Thus a function-maximization process may have reached a stage where most specimens are already close to the maximum. This kind of "degeneration" is certainly *not* "premature." The situation is different if the best chromosome still represents a far-from-perfect solution. In this event, we suspect premature degeneration, which forces us to increase diversity.

Increasing Diversity Several strategies can be followed. The simplest just inserts in the current population some newly created random individuals. A more sophisticated approach will run the genetic algorithm on two or more populations in parallel, in isolation from each other. Then, either at random intervals or whenever premature degeneration is suspected, a specimen from one population will be permitted to choose its mating partner in a different population—the programmer then must not forget to implement a mechanism that decides in which of the two parent populations to place the children. Yet another solution will temporarily increase (significantly) the mutation rate.

Impact of Population Size The size of the population should reflect the needs of the concrete application. As a rule of thumb, small populations need many generations to reach good solutions, and they may prematurely degenerate. Very large populations are robust against degeneration, but they may incur impractical computational costs.

The number of chromosomes is kept constant throughout the algorithm's run, but this is no dogma. We have already seen that, when degeneration is suspected, one possible solution is to enrich the population by additional chromosomes.

21.4.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- In what way does the success of the genetic algorithm depend on the concrete choice of the fitness function? What are the two main pitfalls? How would you handle them?
- What criteria to terminate the run of the genetic algorithm would you recommend? What are their advantages and disadvantages?
- What is *premature degeneration*? How can it be detected and how can the situation be rectified? What is the impact of the population's diversity?
- Discuss the impact of the population size.

21.5 Other Genetic Operators

So far we have considered only the baseline version of the genetic algorithm and its operators. Now that the reader understands the principle, we can take a look at some more sophisticated possibilities.

Two-Point Crossover The one-point crossover from the previous section is a special case of the much more common *two-point crossover*. Here, the random-number generator is asked to return two integers that define two locations in the binary strings. The parents then swap the sub-strings between these two locations as illustrated below (the locations of the crossover points are indicated by spaces).

$$\begin{array}{c} 110\ 110\ 01 \\ 001\ 001\ 11 \end{array} \Rightarrow \begin{array}{c} 110\ 001\ 01 \\ 001\ 110\ 11 \end{array}$$

The crossover points can be different for each chromosome. In this event, each parent will "trade" a different sub-string of its chromosome as indicated below.

$$\begin{array}{l} 1\ 101\ 1001 \\ 001\ 001\ 11 \end{array} \Rightarrow \begin{array}{l} 1\ 001\ 1001 \\ 001\ 101\ 11 \end{array}$$

Random Bit Exchange Yet another variation on the recombination theme is *random bit exchange*. Here, the random-number generator selects a user-specified number of locations, and the genetic algorithm swaps the bits at these locations:

$$\begin{array}{l} 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1 \\ 0\ 0\ 1\ 0\ 0\ 1\ 1\ 1 \end{array} \Rightarrow \begin{array}{l} 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1 \\ 0\ 1\ 1\ 0\ 0\ 0\ 1\ 1 \end{array}$$

The reader has noticed that the second and sixth bits (counting from the left) were swapped. Note that nothing will happen if the leftmost bit is exchanged because it has the same value in both chromosomes. The number of exchanged bits can vary but most applications prefer this to be much less than the chromosome's length.

Common practice combines two or more recombination operators. For instance, the selected pair of parents will with 50% probability be subjected to a 2-point chromosome, with 30% probability to random bit exchange, and with 20% probability there will be no recombination at all.

Inversion While the recombination operators act on pairs of chromosomes, other operators act on individuals. One such operator is mutation; another is *inversion*. In a typical implementation, the random-number generator returns two integers that define two locations in the binary string (as in the 2-point crossover). Then, the sub-string between the two locations is inverted as shown below.

$$110\ 110\ 01 \Rightarrow 110\ 011\ 01$$

Note that the order of the zeros and ones in the sub-string between the third and the seventh bit (counting from the left) was reversed. The location of the two points determines how much inversion the chromosome is impacted. If the two integers are close to each other, say, 4 and 7, only a small part of the chromosome is affected.

Inversion is used to supplement mutation. For instance, the probability that a given bit is mutated can be set to 1% whereas each chromosome may have a 0.5% chance to see its random sub-string inverted. Similarly as with mutation, inversion should be used rarely so as not to destroy the positive contribution of recombination.

Inversion and Premature Degeneration When the program gets trapped in premature degeneration, inversion is better than mutation at extricating it. To see why, consider the following degenerated population.

```
0 1 0 0 0
0 1 0 0 1
0 1 0 0 0
0 1 0 0 0
```

Inverting the middle three bits of the first chromosome, and the last three bits of the second chromosome will result in the following population:

```
0 0 0 1 0
0 1 1 0 0
0 1 0 0 0
0 1 0 0 0
```

The reader can see that the population's diversity has indeed improved. Therefore, when you suspect premature degeneration, just increase, for a while, the frequency of inversions, and perhaps also that of mutations.

21.5.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- Explain the differences between one-point crossover, two-point crossover, and random bit exchange.
- In what specific aspect is recombination different from mutation and inversion?
- How does inversion affect the program's behavior?

21.6 Advanced Versions

The genetic algorithm is a versatile paradigm that allows almost infinite range of variations. This section will introduce some of them.

Abandoning Biological Constraints Computer programs do not have to copy biology. Quite often, the engineer discards some of the living world's limitations in the same way that early aviators shrugged off the need for feathered wings. We have already encountered one such violation when permitting some specimens to become "immortal" by copying them into the new generation alongside with their children. Let us take a look at some others.

Lamarck's Alternative In the baseline version, new chromosomes result only from the application of such operators as recombination or mutation. Throughout the individual's life, its chromosome then remains unchanged. One pre-Darwinian biologist, Jean-Baptiste Lamarck, proposed something more flexible. In his view, evolution might be driven by the individuals' needs: a giraffe that keeps trying to reach topmost foliage will stretch his neck that will thus become longer; and this longer neck is then passed on to the offspring. While untenable in the realm of biology, the hypothesis makes sense elsewhere. Thus a researcher that publishes a scientific paper leaves to posterity the knowledge acquired during her life.

Importantly from our technological perspective, Lamarckian evolution is much faster than the classical Darwinian process; this is why we sometimes implement it in the genetic algorithm. The simplest way to incorporate this concept in the general loop from Fig. 21.1 is to place the "Lamarckian" operator between the "wheel of fortune" and recombination. The task for the operator is to improve the chromosome by adaptation. For instance, one can ask what happens if a certain bit gets flipped-flopped by mutation. Whereas mutation by itself is irreversible, we can add flexibility by testing, beforehand, what happens when any bit is changed, and then choose the best option.

Multi-population Search One motivation for multi-population search has to do with the many parameters that control the genetic algorithm. Most of the time, the engineer has to rely only on her intuition and experience. Alternatively, she may subject the same initial population to several parallel runs of the algorithm, each with its own mutation frequency, with or without inversion, with a different mixture of recombination operators, or with a modified fitness function. Among the many alternatives, some will reach the solution faster than the others.

The attentive reader will recall that multi-population search was also mentioned as a possible solution to the problem of premature degeneration. In that particular context, the idea was to let two or more populations evolve in an isolation punctuated by occasional interbreeding. Note that this interbreeding may not be a straightforward operation if each population represents the individuals by different chromosomes (e.g., by different attributes). The programmer then has to write a special program module for the conversion from one population to another.

Strings of Numbers, Strings of Symbols Chromosomes do not have to be binary strings; they can consist of numbers or symbols just as well. The same recombination operators as before can then be used, though mutation may call for some creativity. Perhaps the most common kind of mutation in numeric strings is to use "noise" superimposed on some (or all) of the chromosome's "genes." For instance, if all locations contain numbers from the interval [0, 100], then the noise can be modeled as a random number from $[-a, a]$ where a is a user-set parameter that plays here a role similar to that of mutation frequency in binary strings. Here is how it can work:

before mutation	10	22	17	42	16
the "noise"		−3	1	−2	
after mutation	10	19	18	40	16

The situation is slightly different if the chromosomes are strings of symbols. Here, mutation can replace a randomly selected symbol in the chromosome with

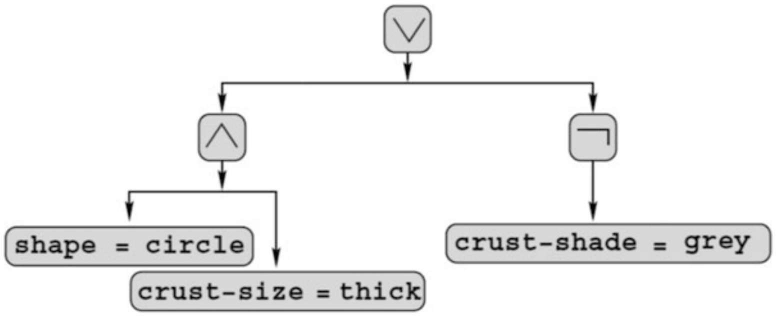


Fig. 21.5 Tree-structure representation of a candidate expression from the "pies" domain from Chap. 1

another symbol chosen by the random-number generator. For instance, when applied to chromosome `[d s r d w k l]`, the mutation can change from `r` to `s` the third symbol from the left, the resulting chromosome being `[d s s d w k l]`.

Also possible are "mixed" chromosomes where some locations are binary, others numeric, and yet others symbolic. Here, mutation usually combines the individual approaches. For instance, the program selects a random location in the chromosome, determines whether the location is binary, numeric, or symbolic, and then applies the appropriate type of mutation.

Chromosomes Implemented as Tree Structures In some applications, strings of bits, numbers, or symbols are inadequate; a tree-structure may prove more appropriate. This, for instance, is the case of classifiers in the form of logical expressions—see the example in Fig. 21.5 where the following expression is represented by a tree-like chromosome.

```
(shape=circle ∧ crust-size=thick) ∨ ¬ crust-shade=gray
```

The expression consists of attributes, the values of these attributes, and the logical operators of conjunction, disjunction, and negation. Note how naturally this is cast in a tree structure whose internal nodes carry out the logical operations and leaves contain attributes and their values. Recombination swaps random subtrees. Mutation can affect the leaves: either attribute names or attribute values or both. Another possibility for mutation is occasionally to replace $\wedge$ with $\vee$ or the other way around.

Special attention has to be devoted to the way the initial population is generated. The programmer should make sure that the population already contains some promising logical expressions. One possibility is to create a group of random expressions and to insert in them parts of the descriptions of the positive examples. The survival function (to be maximized) can be defined as the classification accuracy on the training set.

21.6.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What is the difference between the Darwinian and Lamarckian evolution processes? Which of them is faster?
- What weakness is remedied by the multi-population genetic algorithm? In what way do multiple populations address this weakness?
- How would you implement the mutation operator if the chromosome is a "mixed" string of bits, numbers, and symbols?
- How would you implement the recombination and mutation operators in domains where chromosomes have the form of tree data structures?

21.7 Choices Made by *k*-NN Classifiers

Let us discuss a possible application of the genetic algorithm to a concrete machine-learning problem.

Attribute Selection and Example Selection Recall that the success of the k-NN classifier depends on the quality of the stored examples (some can be noisy or less representative) and also on the choice of the attributes to describe these examples (some can be redundant or irrelevant).

The problem of choosing the right examples and attributes is easily addressed by the genetic algorithm. To be able to use it, however, we have to decide how to represent the problem in terms of chromosomes, how to define the fitness function, and how to implement the recombination and mutation operators. We also have to be clear about how to interpret (and utilize) the results of the genetic search.

Chromosomes to Encode the Problem A very simple approach will divide the binary chromosome into two parts: each bit in the first part corresponds to one training example, and each bit in the second part corresponds to one attribute (Fig. 21.6). If the value of a certain bit is `0`, the corresponding example or attribute is ignored; if it is `1`, the example or attribute is employed. The fitness function is designed in a way that seeks to maximize classification performance with the minimum number of 1's in the chromosomes (i.e., the minimum number of examples and the minimum number of attributes).

This approach may require impractically long chromosomes in domains where the training set has many examples and many attributes: if the training set consists of ten thousand examples, ten thousand bits are needed. A better solution will opt for the more flexible variable-length scheme where each element in the chromosome contains an integer that points to a training example or an attribute. The length of the chromosome equals the number of relevant attributes plus the number of representative examples. The mechanism is known as *value encoding*.

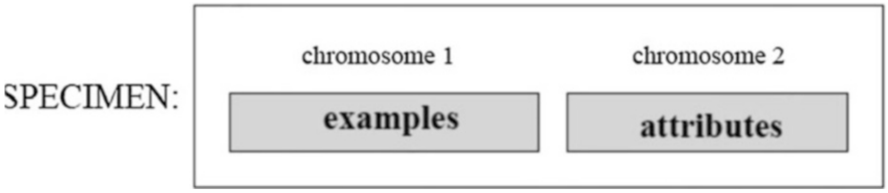


Fig. 21.6 Each individual is described by two chromosomes, one for examples; the other for attributes. Recombination is applied to each of them separately

Interpreting the Chromosomes We must interpret these pairs of chromosomes correctly. For instance, the specimen `[3,14,39]`, `[2,4]` represents a training subset consisting of the third, fourteenth, and thirty-ninth training examples, described by the second and fourth attributes. When such specimen is used to define a classifier, the program selects the examples determined by the first chromosome and describes them by the attributes pointed to by the second chromosome. The distances between vectors $\mathbf{x} = (x_1, \ldots x_n)$ and $\mathbf{y} = (y_1, \ldots, y_n)$ are calculated using the formula:

$$D(\mathbf{x}, \mathbf{y}) = \sqrt{\Sigma_{i=1}^{n} d(x_i, y_i)} \tag{21.1}$$

where $d(x_i, y_i)$ is the contribution of the ith attribute (of course only the attributes found in the chromosome are employed). For numeric attributes, this contribution is $d(x_i, y_i) = (x_i - y_i)^2$; for Boolean attributes and for discrete attributes, we may define $d(x_i, y_i) = 0$ if $x_i = y_i$ and $d(x_i, y_i) = 1$ if $x_i \neq y_i$.

Fitness Function The next question is how to determine each individual's survival chances. Recall that we want to reduce the number of examples and attributes without compromising classification accuracy. These requirements may conflict with each other because, in noise-free domains, the entire training set tends to give higher classification performance than a reduced set. Likewise, removing attributes is hardly beneficial if each of them provides relevant information.

The involved trade-offs ought to be reflected in fitness-function parameters that allow the user to specify his or her preferences—to place emphasis either on maximizing classification accuracy or on minimizing the number of the retained training examples and attributes. One way of doing so is by the following formula where E_R is the number of training examples misclassified by the given specimen, N_E is the number of retained examples, and N_A is the number of retained attributes:

$$f = \frac{1}{c_1 * E_R + c_2 * N_E + c_3 * N_A} \tag{21.2}$$

Note that the fitness of a specimen is high if its error rate is low, if the set of retained examples is small, and if many attributes have been eliminated. The function is controlled by three parameters, c_1, c_2, and c_3, that reflect the user's preferences. For instance, if c_1 is high, emphasis is placed on classification accuracy. If c_2 or c_3 are high, emphasis is placed on minimizing the numbers of retained examples and attributes.

Genetic Operators Employed in This Application Parents are selected probabilistically. Specifically, the following formula is used to calculate the probability that the specimen S' will be chosen:

$$Prob(S') = \frac{f(S')}{\sum_S f(S)} \tag{21.3}$$

Here, $f(S)$ is the fitness of specimen S as calculated by Eq. (21.2). The denominator sums the fitness values of all specimens in the population so as to make the probabilities sum to 1.

Once the pair of parents have been chosen, their chromosomes are recombined by the two-point crossover. Since each specimen is defined by a pair of chromosomes, each with a different interpretation, we apply the recombination operator to each of them separately.

Let the length of one parent's chromosome be denoted by N_1 and let the length of the other parent's chromosome be denoted by N_2. Using uniform distribution, the algorithm selects one pair of integers from the closed interval $[1, N_1]$ and another pair of integers from the closed interval $[1, N_2]$. Each of these pairs then defines a sub-string in the respective chromosome (the first and the last locations are included in the sub-string). The crossover operator then exchanges the sub-strings from one of the parent chromosomes with the sub-strings of the other parent. Note that, as each of these sub-strings can have a different size, the children's lengths are likely to be different from the parents' lengths.

Graphical Illustration The principle is illustrated by Fig. 21.7 where the middle parts of chromosomes A and B have been exchanged. Note how the lengths of A and B have thus been affected. The engineer has to decide whether to permit the situation where the exchanged segments have size 0; at the other extreme, a segment can represent the entire parent.

The *mutation* operator should prevent premature degeneration and make sure the population adequately represents the solution space. One possibility it to select, randomly, a pre-specified percentage of the locations in the chromosomes

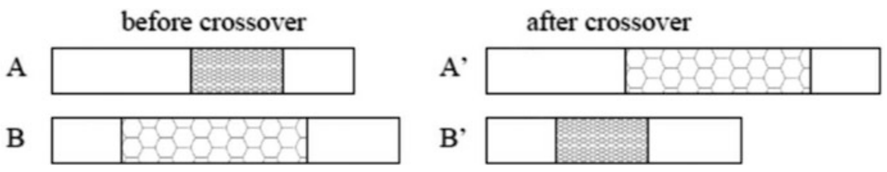


Fig. 21.7 Two-point crossover operator creates the children by exchanging randomly selected sub-strings in the parent chromosomes

of the newly created population and to add to each of them a random integer generated separately for the location. The result is then taken modulo the number of examples/attributes. Let the original number of examples/attributes be 100 and let the location selected for mutation contains be 95. If the randomly generated integer is 22, then the value after mutation is $(95 + 22) \bmod 100 = 17$.

21.7.1 What Have You Learned?

To make sure you understand this topic, try to answer the following questions. If you have problems, return to the corresponding place in the preceding text.

- What can be accomplished by choosing the best attributes and the most representative examples in k-NN classifiers?
- What is the motivation behind the suggestion to work with two chromosomes instead of just one?
- How does the chosen fitness function reflect the competing requirements of small sets of attributes and examples versus high classification accuracy?
- Why did we use a recombination operator that exchanges sub-strings of different lengths? How was mutation implemented?

21.8 Summary and Historical Remarks

- The genetic algorithm, inspired by Darwinian evolution, is a useful took for optimization and search for acceptable solutions to difficult engineering problems. The simplest implementation works with chromosomes in the form of binary strings.
- The algorithm works with a population of individuals represented by the chromosomes. A principal assumption is the availability of a function capable of returning for each chromosome its *fitness*, understood as ability to survive in Darwinian competition.
- The population of individuals is subjected to three essential operations: fitness-based survival, recombination of pairs of chromosomes, and mutation. Also inversion of a sub-string is sometimes used.
- One frequently encountered problem in practical applications is the population's *premature degeneration*, which can be detected by reduced diversity of the population. One solution will add artificially created chromosomes to the population. Other possibilities include the inversion operator and increased frequency of mutations.
- Alternative implementations of the genetic algorithm use strings of numbers, symbols, mixed strings, or even tree structures. Sometimes, the individuals are

described by two or more chromosomes. An interesting idea is to use multiple populations with interbreeding.
- The chapter illustrated the practical use of the genetic algorithm on a simple task from the field of nearest-neighbor classifiers. In this problem, three competing performance criteria had to be satisfied.

Historical Remarks The idea to describe biological evolution in terms of computer algorithm is due to Holland (1975), although some other authors suggested something similar a little earlier. Among these, perhaps Rechenberg (1973) deserves to be mentioned, while Fogel et al. (1966) should be credited with pioneering the idea of genetic programming. The concrete way of applying the genetic algorithm to selections in the k-NN classifier is from Rozsypal and Kubat (2001).

21.9 Solidify Your Knowledge

The exercises are to solidify the acquired knowledge. The suggested thought experiments will help the reader see this chapter's ideas in a different light and provoke independent thinking. Computer assignments will force the readers to pay attention to seemingly insignificant details they might otherwise overlook.

21.9.1 Exercises

1. Hand-simulate the genetic algorithm with a pencil and paper in a similar way as in Table 21.2. Use a fitness function of your own choice, a different initial population, and random locations for the one-point crossover. Repeat the exercise with the two-point crossover.

21.9.2 Give It Some Thought

1. Explain how different population sizes may affect the number of generations needed to reach a good solution. Elaborate on the relation of population size to the problem of premature degeneration. Discuss also the effect of the shape of the fitness function.
2. Identify concrete engineering problems (other than those in this book) appropriate for the genetic algorithm. Suggest problems where the chromosomes are best represented by binary or numeric strings, and suggest problems where trees are more natural.
3. Name some differences between natural evolution and its computer model. Speculate on whether more inspiration can be derived from nature. Where do

you think are the advantages of the computer programs as compared to biological evolution?

4. Suggest a mechanism that would use the genetic algorithm in the search for a good architecture of multilayer perceptrons (consider MLPs with more than one hidden layer).

21.9.3 Computer Assignments

1. Implement the baseline genetic algorithm that operates on binary-string chromosomes. Make sure you have written separate modules for the survival function, the wheel of fortune, recombination, and mutation. Make sure these modules allow easy modifications.
2. Create an initial population for the "pies" domain from Chap. 1 and use it as input to the program developed in the previous task.
3. For a domain of your own choice, implement a few alternative mating strategies. Run systematic experiments to decide which strategy will be the fastest in finding the solution. The speed can be measured by the number of chromosomes whose fitness values have been evaluated before the solution is found.
4. For a domain of your own choice, experiment with alternative "cocktails" of different recombination operators, and with different frequencies of recombination, mutation, and inversion. Plot graphs that show how the speed of search (measured as in the previous task) depends of the concrete settings of these parameters.
5. Implement the genetic algorithm working with two or more populations that occasionally interbreed.

Bibliography

Ash, T. (1989). Dynamic node creation in backpropagation neural networks. *Connection Science: Journal of Neural Computing, Artificial Intelligence, and Cognitive Research, 1*, 365–375.

Baker, J. (1975). The DRAGON system–An overview. *IEEE Transactions on Acoustics, Speech, and Signal Processing, 23*, 24–29.

Ball, G. H., & Hall, D. J. (1965). ISODATA, A Novel Method of Data Analysis and Classification. Technical Report of the Stanford University, Stanford, CA.

Bartlett, P. L., Harvey, N., Liaw, C., & Mehrabian, A. (2019). Nearly-tight VC-dimension and pseudo-dimension bound for piece-wise linear neural networks. *Journal of Machine Learning Research, 20*(63), 1–17.

Baum, L. E., & Petrie, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Mathematical Statistics, 37*(6), 1554–1563.

Bellman, R. E. (1956). A problem in the sequential design of experiments. *Sankhya, 16*, 221–229.

Bellman, R. E. (1957). *Dynamic programming.* Princeton: Princeton University Press.

Blake, C. L., & Merz, C. J. (1998). *Repository of Machine Learning Databases*. Department of Information and Computer Science, University of California at Irvine, www.ics.uci.edu/~mlearn/MLRepository.html

Blumer, W., Ehrenfeucht, A., Haussler, D., & Warmuth, M. K. (1989). Learnability and the Vapnik-Chervonenkis dimension. *Journals of the ACM, 36*, 929–965.

Boutell, M. R., Luo, J., Shen, X., & Brown, C. M. (2004). Learning multi-label scene classification. *Pattern Recognition, 37*, 1757–1771.

Bower, G. H., & Hilgard, E. R. (1981). *Theories of learning*. Englewood Cliffs: Prentice-Hall.

Breiman, L. (1996). Bagging predictors. *Machine Learning, 24*, 123–140.

Breiman, L. (2001). Random forests. *Machine Learning, 45*, 5–32.

Breiman, L., Friedman, J., Olshen, R., & Stone, C. J. (1984). *Classification and regression trees*. Belmont: Wadsworth International Group.

Broomhead, D. S. & Lowe, D. (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems, 2*, 321–355.

Bryson, A. E. & Ho, Y.-C. (1969). *Applied optimal control*, New York: Blaisdell.

Charniak, E. (2018). *Introduction to deep learning*. Cambridge: The MIT Press.

Chow, C. K. (1957). An optimum character recognition system using decision functions. *IRE Transactions on Computers, EC-6*, 247–254.

Clare, A., & King, R. D. (2001). Knowledge discovery in multi-label phenotype data. In *Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery, PKDD'01*, Freiburg (pp. 42–53)

Clark, P., & Niblett, R. (1989). The CN2 induction algorithm. *Machine Learning, 3*, 261–284.

Coppin, B. (2004). *Artificial intelligence illuminated.* Sudbury: Jones and Bartlett.

Cover, T. M. (1965). Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers, EC-14*, 326–334.

Cover, T.M. (1968). Estimation by the nearest neighbor rule. *IEEE Transactions on Information Theory, IT-14*, 50–55.

Cover, T. M., & Hart, P. E. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory, IT-13*, 21–27.

Dasarathy, B. V. (1991). *Nearest-neighbor classification techniques*. Los Alomitos: IEEE Computer Society Press.

Dietterich, T. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation, 10*, 1895–1923.

Dudani, S. A. (1975). The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics, SMC-6*, 325–327.

Elman, J. (1990). Finding structure in time. *Cognitive Science, 14*(2), 179–211.

Fayyad, U. M., & Irani, K. B. (1992). On the handling of continuous-valued attributes in decision tree generation. *Machine Learning, 8*, 87–102.

Fisher, R. A. (1936). The use of multiple measurement in taxonomic problems. *Annals of Eugenics, 7*, 111–132.

Fisher, D. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning, 2*, 139–172.

Fix, E., & Hodges, J. L. (1951). *Discriminatory Analysis, Non-parametric Discrimination*. USAF School of Aviation Medicine, Randolph Field, TX, Project 21-49-004, Report 4, Contract AF41(128)-3.

Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial intelligence through simulated evolution*. New York: Wiley.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. *Machine Learning: Proceedings of the Thirteenth International Conference*, Bari (pp. 148–156).

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software, 3*(3), 209–226.

Gennari, J. H., Langley, P., & Fisher, D. (1990). Models of incremental concept formation. *Artificial Intelligence, 40*, 11–61.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Sardinia, May 13–15 (pp. 249–256)

Godbole, S., & Sarawagi, S. (2004). Discriminative methods for multi-label classification. In H. Dai, R. Srikant, & C. Zhang (Eds.), *Lecture Notes in Artificial Intelligence* (Vol. 3056, pp 22–30). Berlin/Heidelberg: Springer.

Good, I. J. (1965). *The estimation of probabilities: An essay on modern Bayesian methods*. Cambridge: MIT.

Gordon, D. F., & desJardin, M. (1995). Evaluation and selection of biases in machine learning. *Machine Learning, 20*, 5–22.

Hart, P. E. (1968). The condensed nearest neighbor rule. *IEEE Transactions on Information Theory, IT-14*, 515–516.

Hellman, M. E. (1970). The nearest neighbor classification rule with the reject option. *IEEE Transactions on Systems Science and Cybernetics, 6*, 179–185.

Hinton, G. E., & Zemel, R. S. (1994). Auto-encoders, minimum description length, and Helmholz free energy. In *Advances in neural information processing systems* (Vol. 6, pp. 3–10).

Hochreiter, S., & Schmidhuber, J. (1997). Long short term memory. *Neural computation, 9* (8), 1735–1780.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press.

Holte, R. C. (1993). Very simple classification rules perform well on most commonly used databases. *Machine Learning, 11*, 63–90.

Hunt, E. B., Marin, J., & Stone, P. J. (1966). *Experiments in induction*, New York: Academic Press.

Katz, A. J., Gately, M. T., & Collins, D. R. (1990). Robust classifiers without robust features. *Neural Computation, 2*, 472–479.

Kearns, M. J., & Vazirani, U. V. (1994). *An introduction to computational learning theory*. Cambridge, MA: MIT Press.

Kodratoff, Y. (1988). *Introduction to machine learning*, London: Pitman.

Kodratoff, Y., & Michalski, R. S. (1990). *Machine learning: An artificial intelligence approach* (Vol.3). San Mateo: Morgan Kaufmann.

Kohavi, R. (1997). Wrappers for feature selection. *Artificial Intelligence, 97*(1–2), 273–324.

Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics, 43*, 59–69.

Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE, 78*(9), 1464–1480.

Koller, D., & Sahami, M. (1997). Hierarchically classifying documents using very few words. In *Proceedings of the 14th International Conference on Machine Learning, ICML'07*, San Francisco (pp. 170–178)

Kononenko, I., Bratko, I., & Kukar., M. (1998). Application of machine learning to medical diagnosis. In: R. Michalski, I. Bratko, & M. Kubat (Eds.), *Machine learning and data mining: Methods and applications*. Chichester: Wiley.

Krizhevsky, A., Sutskever, I., & Hinton G. (2012). Imagenet classification with deep convolutional neural network. In *Advances in neural information processing systems* (pp. 1097–1105).

Kubat, M. (1989). Floating approximation in time-varying knowledge bases. *Pattern Recognition Letters, 10*, 223–227.

Kubat, M., Holte, R., & Matwin, S. (1997). Learning when negatives examples abound. In *Proceedings of the European Conference on Machine Learning (ECML'97)*, April 1997, Prague (pp. 146–153).

Kubat, M., Holte, R., & Matwin, S. (1998). Detection of oil-spills in radar images of sea surface. *Machine Learning, 30*, 195–215.

Kubat, M., Koprinska, I., & Pfurtscheller, G. (1998). Learning to classify medical signals. In R. Michalski, I. Bratko, & M. Kubat (Eds), *Machine learning and data mining: Methods and applications*. Chichestser: Wiley.

Kubat, M., Pfurtscheller, G., & Flotzinger D. (1994). AI-based approach to automatic sleep classification. *Biological Cybernetics, 79*, 443–448.

LeCun, Y. (1987). *Modèles Connexionnistes de l'apprentissage*. PdD dissertation, University of Paris.

LeCun,Y., Boser, B. E., Denker, J. S. Henderson, D., Howard, R. E., Hubbard, W. E., & Jackel, L. D. (1989). Handwritten digit recognition with a backpropagation network. In *Advances in neural information processing systems* (pp. 396–404)

Lewis, D. D., & Gale, W. A. (1994). A sequential algorithm for training text classifiers. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'94)*, Dublin (pp. 3–12).

Littlestone, N. (1987). Learning quickly when irrelevant attributes abound: A new linear threshold algorithm. *Machine Learning, 2*, 285–318.

Louizou, G., & Maybank, S. J. (1987). The nearest neighbor and the bayes error rates. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 9*, 254–262.

McCallum, A. (1999). Multi-label text classification with a mixture model trained by EM. In *Proceedings of the workshop on text learning (AAAI'99)* (pp. 1–7).

McQueen, J. (1967). Some methods for classification and analysis of multivariate observations. *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley (pp. 281–297).

Michalski, R. S. (1969). On the quasi-minimal solution of the general covering problem. In *Proceedings of the 5th International Symposium on Information Processing (FCIP'69)*, Bled, Yugoslavia (Vol. A3, pp. 125–128).

Michalski, R., Bratko, I., & Kubat, M. (1998). *Machine learning and data mining: Methods and applications*. New York: Wiley.

Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (1983). *Machine learning: An artificial intelligence approach*. Palo Alto: Tioga Publishing Company.

Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (1986). *Machine learning: An artificial intelligence approach* (Vol. 2). Palo Alto: Tioga Publishing Company.

Michalski, R. S. & Tecuci, G. (1994). *Machine learning: A multistrategy approach* Palo Alto: Morgan Kaufmann.

Mill, J.S. (1865). *A system of logic*. London: Longmans.

Minsky, M., & Papert, S. (1969). *Perceptrons*. Cambridge, MA: MIT.

Mitchell, M. (1998). *An introduction to genetic algorithm*. Cambridge, MA: MIT.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence, 18*, 203–226.

Mitchell, T. M. (1997). *Machine learning*. New York: McGraw-Hill.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature, 518*(7540), 529–533.

Mori, S, Suen, C. Y., & Yamamoto, K. (1992). Historical overview of OCR research and development. *Proceedings of IEEE, 80*, 1029–1058.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the 5th International Machine Learning Conference*, Ann Arbor, Michigan (pp. 339–352).

Murty, M. N., & Krishna, G. (1980). A computationally efficient technique for data clustering. *Pattern Recognition, 12*, 153–158.

Neyman, J., & Pearson E. S. (1928). On the use and interpretation of certain test criteria for purposes of statistical inference. *Biometrica, 20A*, 175–240.

Ogden, C. K., & Richards, I. A. (1923). *The meaning of meaning*. New York: Harcourt, Brace, and World. Eighth edition 1946.

Parzen E. (1962). On estimation of a probability density function and mode. *Annals of Mathematical Statistics, 33*, 1065–1076.

Quinlan, J. R. (1979). Discovering rules by induction from large collections of examples. In D. Michie (Ed.), *Expert systems in the micro electronic age*. Edinburgh: Edinburgh University Press.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*, 81–106.

Quinlan, R. (1990). Learning logical definitions from relations. *Machine Learning, 5*, 239–266.

Quinlan, J. R. (1993). *C4.5: Programms for machine learning.* San Mateo: Morgan Kaufmann.

Read, J, Pfahringer, B., Holmes, G., & Frank, E. (2011). Classifier chains for multi-label classification. *Machine Learning, 85*, 333–359.

Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Principien der biologischen Evolution*. Stuttgart: Frommann-Holzboog.

Rosenblatt, M. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review, 65*, 386–408.

Rozsypal, A. R., & Kubat, M. (2001). Using the genetic algorithm to reduce the size of a nearest-neighbor classifier and to select relevant attributes. In *Proceedings of the 18th International Conference on Machine Learning*, Williamstown (pp. 449–456).

Rumelhart, D. E., Hinton, G. E., & Williams, R. J., (1986). Learning representations by backpropagating errors. *Nature, 323*, 533–536.

Rumelhart, D. E. & McClelland, J. L. (1986). *Parallel distributed processing*. Cambridge: MIT Bradford Press.

Rummery, G. A., & Niranjan, M. (1994). *On-line Q-learning Using Connectionist Systems*. Technical Report CUED/F-INFENG/TR 166, Engineering Department, Cambridge University.

Russell, S., & Norvig, P. (2003). *Artificial intelligence, a modern approach* (2nd ed.). Englewood Cliffs: Prentice Hall.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning, 5*, 197–227.

Shawe-Taylor, J., Anthony, M., & Biggs, N. (1993). Bounding sample size with the Vapnik-Chervonenkis dimension. *Discrete Applied Mathematics, 42*(1), 65–73.

Stamp, M. (2018), *Introduction to machine learning with applications in information security*. Boca Raton, FL: CRC Press, Taylor & Francis Group.

Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning*. PhD Dissertation, University of Massachusetts, Amherst.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning, 3*, 9–44.

Sutton, R. S., & Barto, A. G. (1998): *Reinforcement learning: An introduction*. Cambridge: MIT Press.

Thrun, S. B. & Mitchell, T. M. (1995). Lifelong robot learning. *Robotics and Automonous Systems, 15*, pp. 24–46.

Tomek, I. (1976). Two modifications of CNN. *IEEE Transactions on Systems, Man and Communications, SMC-6*, 769–772.

Turney, P. D. (1993). Robust classification with context-sensitive features. *Proceedings of the Sixth International Conference of Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Edinburgh (pp.268–276).

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM, 27*, 1134–1142.

Vapnik, V. N. (1992). *Estimation of dependences based on empirical data*. New York: Springer.

Vapnik, V. N. (1995). *The nature of statistical learning theory.* New York: Springer.

Vapnik, V. N., & Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications, 16*, 264–280.

Watkins, C. J. C. H, & Dayan P. (1992). Q-learning. *Machine Learning, 8*, 279–292.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University.

Whewel, W. (1858). *History of scientific ideas*. London: J.W. Parker.

Widmer, G. (1997). Tracking context changes through meta-learning. *Machine Learning, 27*, 259–286.

Widmer, G., & Kubat, M. (1996). Learning in the presence of concept drift and hidden contexts. *Machine Learning, 23*, 69–101.

Widrow, B., & Hoff, M. E. (1960). Adaptive switching circuits. In *IRE WESCON convention record*, New York (pp. 96–104).

Wolpert, D. (1992). Stacked generalization. *Neural Networks, 5*, 241–259.

Wolpert, D. (1996). The lack of a priori distinctions between learning algorithms. *Neural Computation, 8*, 1341–1390.

Zhang, M.-L., & Zhou, Z.-H. (2007). ML-KNN: A lazy learing approach to multi-label learning. *Pattern Recognition, 40*, 2038–2048.

Index