**Machine Learning Guidelines**

| S.no. | | |
|---|---|---|
| 1 | Basic definitions | Reference [2], Chapter 1, page 1-5 |
| 2 | Hypothesis space and inductive bias | Reference [2], Chapter 2, section 2.1 - 2.4 |
| 3 | Bayes optimal classifier and Bayes error, Naive Bayes classifier | Reference [2], Chapter 6, section 6.1, 6.2, 6.7, 6.9 |
| 4 | Curse of dimensionality, dimensionality reduction, feature scaling, feature selection methods | Reference[3] Page 33-35 Reference [1], Chapter 10, |
| 5 | Linear regression with one variable, linear regression with multiple variables | Reference [1], Chapter 7, page 194-205 |
| 6 | Gradient Descent | https://medium.com/analytics-vidhya/linear-regression-with-gradient-descent-derivation-c10685ddf0f4 |
| 7 | logistic regression, over-fitting, regularization. performance evaluation metrics, validation methods | 1. https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148<br>2. https://medium.com/@shiny_jay/ml-regularization-79a081666fbc<br>3. https://medium.com/@qempsil0914/courseras-machine-learning-notes-week3-overfitting-and-regularization-partii-3e3f3f36a287 |
| 8 | Decision trees | Reference [2], ~~Chapter 8,~~ page 52 - 60, 63 - 66 Chapter 3 |
| 9 | k-nearest neighbor classifier | Reference [2], Chapter 8, page 231-233 |
| 10 | perceptron, multilayer perceptron, neural networks, back-propagation algorithm | Reference [2], Chapter 4, page 81-99 |
| 11 | Support Vector Machine (SVM), Kernel functions | Reference [1], Chapter 7, section 7.3<br>Reference [3], Chapter 6, page 292 |

| | | |
|---|---|---|
| 12 | Approaches for clustering, distance metrics, K-means clustering, expectation maximization, hierarchical clustering | Reference [3], Chapter 9, page 424-426 Reference [1], Chapter 8, section 8.1, 8.5 |
| 13 | performance evaluation metrics | https://www.kdnuggets.com/2020/05/model-evaluation-metrics-machine-learning.html |
| 14 | validation methods | https://www.analyticsvidhya.com/blog/2021/05/4-ways-to-evaluate-your-machine-learning-model-cross-validation-techniques-with-python-code/ |

**References:**

1. Flach, P. (2015). Machine Learning: The Art and Science of Algorithms that Make Sense of Data. Cambridge University Press.

2. Mitchell, T.M. (2017). Machine Learning. McGraw Hill Education.

**Additional References:**

3. Christopher & Bishop, M. (2016). Pattern Recognition and Machine Learning. New York: Springer-Verlag Sense of Data. Cambridge University Press.

# CHAPTER 1

# INTRODUCTION

Ever since computers were invented, we have wondered whether they might be made to learn. If we could understand how to program them to learn—to improve automatically with experience—the impact would be dramatic. Imagine computers learning from medical records which treatments are most effective for new diseases, houses learning from experience to optimize energy costs based on the particular usage patterns of their occupants, or personal software assistants learning the evolving interests of their users in order to highlight especially relevant stories from the online morning newspaper. A successful understanding of how to make computers learn would open up many new uses of computers and new levels of competence and customization. And a detailed understanding of information-processing algorithms for machine learning might lead to a better understanding of human learning abilities (and disabilities) as well.

We do not yet know how to make computers learn nearly as well as people learn. However, algorithms have been invented that are effective for certain types of learning tasks, and a theoretical understanding of learning is beginning to emerge. Many practical computer programs have been developed to exhibit useful types of learning, and significant commercial applications have begun to appear. For problems such as speech recognition, algorithms based on machine learning outperform all other approaches that have been attempted to date. In the field known as data mining, machine learning algorithms are being used routinely to discover valuable knowledge from large commercial databases containing equipment maintenance records, loan applications, financial transactions, medical records, and the like. As our understanding of computers continues to mature, it

seems inevitable that machine learning will play an increasingly central role in computer science and computer technology.

A few specific achievements provide a glimpse of the state of the art: programs have been developed that successfully learn to recognize spoken words (Waibel 1989; Lee 1989), predict recovery rates of pneumonia patients (Cooper et al. 1997), detect fraudulent use of credit cards, drive autonomous vehicles on public highways (Pomerleau 1989), and play games such as backgammon at levels approaching the performance of human world champions (Tesauro 1992, 1995). Theoretical results have been developed that characterize the fundamental relationship among the number of training examples observed, the number of hypotheses under consideration, and the expected error in learned hypotheses. We are beginning to obtain initial models of human and animal learning and to understand their relationship to learning algorithms developed for computers (e.g., Laird et al. 1986; Anderson 1991; Qin et al. 1992; Chi and Bassock 1989; Ahn and Brewer 1993). In applications, algorithms, theory, and studies of biological systems, the rate of progress has increased significantly over the past decade. Several recent applications of machine learning are summarized in Table 1.1. Langley and Simon (1995) and Rumelhart et al. (1994) survey additional applications of machine learning.

This book presents the field of machine learning, describing a variety of learning paradigms, algorithms, theoretical results, and applications. Machine learning is inherently a multidisciplinary field. It draws on results from artificial intelligence, probability and statistics, computational complexity theory, control theory, information theory, philosophy, psychology, neurobiology, and other fields. Table 1.2 summarizes key ideas from each of these fields that impact the field of machine learning. While the material in this book is based on results from many diverse fields, the reader need not be an expert in any of them. Key ideas are presented from these fields using a nonspecialist's vocabulary, with unfamiliar terms and concepts introduced as the need arises.

## 1.1 WELL-POSED LEARNING PROBLEMS

Let us begin our study of machine learning by considering a few learning tasks. For the purposes of this book we will define learning broadly, to include any computer program that improves its performance at some task through experience. Put more precisely,

> ***Definition***: A computer program is said to **learn** from experience $E$ with respect to some class of tasks $T$ and performance measure $P$, if its performance at tasks in $T$, as measured by $P$, improves with experience $E$.

For example, a computer program that learns to play checkers might improve its performance *as measured by its ability to win* at the class of tasks involving *playing checkers games*, through experience *obtained by playing games against itself*. In general, to have a well-defined learning problem, we must identity these

- Learning to recognize spoken words.

  All of the most successful speech recognition systems employ machine learning in some form. For example, the SPHINX system (e.g., Lee 1989) learns speaker-specific strategies for recognizing the primitive sounds (phonemes) and words from the observed speech signal. Neural network learning methods (e.g., Waibel et al. 1989) and methods for learning hidden Markov models (e.g., Lee 1989) are effective for automatically customizing to individual speakers, vocabularies, microphone characteristics, background noise, etc. Similar techniques have potential applications in many signal-interpretation problems.
- Learning to drive an autonomous vehicle.

  Machine learning methods have been used to train computer-controlled vehicles to steer correctly when driving on a variety of road types. For example, the ALVINN system (Pomerleau 1989) has used its learned strategies to drive unassisted at 70 miles per hour for 90 miles on public highways among other cars. Similar techniques have possible applications in many sensor-based control problems.
- Learning to classify new astronomical structures.

  Machine learning methods have been applied to a variety of large databases to learn general regularities implicit in the data. For example, decision tree learning algorithms have been used by NASA to learn how to classify celestial objects from the second Palomar Observatory Sky Survey (Fayyad et al. 1995). This system is now used to automatically classify all objects in the Sky Survey, which consists of three terrabytes of image data.
- Learning to play world-class backgammon.

  The most successful computer programs for playing games such as backgammon are based on machine learning algorithms. For example, the world's top computer program for backgammon, TD-GAMMON (Tesauro 1992, 1995), learned its strategy by playing over one million practice games against itself. It now plays at a level competitive with the human world champion. Similar techniques have applications in many practical problems where very large search spaces must be examined efficiently.

**TABLE 1.1**
Some successful applications of machine learning.

three features: the class of tasks, the measure of performance to be improved, and the source of experience.

### A checkers learning problem:

- Task $T$: playing checkers
- Performance measure $P$: percent of games won against opponents
- Training experience $E$: playing practice games against itself

We can specify many learning problems in this fashion, such as learning to recognize handwritten words, or learning to drive a robotic automobile autonomously.

### A handwriting recognition learning problem:

- Task $T$: recognizing and classifying handwritten words within images
- Performance measure $P$: percent of words correctly classified

- Artificial intelligence

  Learning symbolic representations of concepts. Machine learning as a search problem. Learning as an approach to improving problem solving. Using prior knowledge together with training data to guide learning.
- Bayesian methods

  Bayes' theorem as the basis for calculating probabilities of hypotheses. The naive Bayes classifier. Algorithms for estimating values of unobserved variables.
- Computational complexity theory

  Theoretical bounds on the inherent complexity of different learning tasks, measured in terms of the computational effort, number of training examples, number of mistakes, etc. required in order to learn.
- Control theory

  Procedures that learn to control processes in order to optimize predefined objectives and that learn to predict the next state of the process they are controlling.
- Information theory

  Measures of entropy and information content. Minimum description length approaches to learning. Optimal codes and their relationship to optimal training sequences for encoding a hypothesis.
- Philosophy

  Occam's razor, suggesting that the simplest hypothesis is the best. Analysis of the justification for generalizing beyond observed data.
- Psychology and neurobiology

  The power law of practice, which states that over a very broad range of learning problems, people's response time improves with practice according to a power law. Neurobiological studies motivating artificial neural network models of learning.
- Statistics

  Characterization of errors (e.g., bias and variance) that occur when estimating the accuracy of a hypothesis based on a limited sample of data. Confidence intervals, statistical tests.

**TABLE 1.2**
Some disciplines and examples of their influence on machine learning.

- Training experience $E$: a database of handwritten words with given classifications

**A robot driving learning problem:**

- Task $T$: driving on public four-lane highways using vision sensors
- Performance measure $P$: average distance traveled before an error (as judged by human overseer)
- Training experience $E$: a sequence of images and steering commands recorded while observing a human driver

Our definition of learning is broad enough to include most tasks that we would conventionally call "learning" tasks, as we use the word in everyday language. It is also broad enough to encompass computer programs that improve from experience in quite straightforward ways. For example, a database system

that allows users to update data entries would fit our definition of a learning system: it improves its performance at answering database queries, based on the experience gained from database updates. Rather than worry about whether this type of activity falls under the usual informal conversational meaning of the word "learning," we will simply adopt our technical definition of the class of programs that improve through experience. Within this class we will find many types of problems that require more or less sophisticated solutions. Our concern here is not to analyze the meaning of the English word "learning" as it is used in everyday language. Instead, our goal is to define precisely a class of problems that encompasses interesting forms of learning, to explore algorithms that solve such problems, and to understand the fundamental structure of learning problems and processes.

## 1.2 DESIGNING A LEARNING SYSTEM

In order to illustrate some of the basic design issues and approaches to machine learning, let us consider designing a program to learn to play checkers, with the goal of entering it in the world checkers tournament. We adopt the obvious performance measure: the percent of games it wins in this world tournament.

### 1.2.1 Choosing the Training Experience

The first design choice we face is to choose the type of training experience from which our system will learn. The type of training experience available can have a significant impact on success or failure of the learner. One key attribute is whether the training experience provides direct or indirect feedback regarding the choices made by the performance system. For example, in learning to play checkers, the system might learn from *direct* training examples consisting of individual checkers board states and the correct move for each. Alternatively, it might have available only *indirect* information consisting of the move sequences and final outcomes of various games played. In this later case, information about the correctness of specific moves early in the game must be inferred indirectly from the fact that the game was eventually won or lost. Here the learner faces an additional problem of *credit assignment*, or determining the degree to which each move in the sequence deserves credit or blame for the final outcome. Credit assignment can be a particularly difficult problem because the game can be lost even when early moves are optimal, if these are followed later by poor moves. Hence, learning from direct training feedback is typically easier than learning from indirect feedback.

A second important attribute of the training experience is the degree to which the learner controls the sequence of training examples. For example, the learner might rely on the teacher to select informative board states and to provide the correct move for each. Alternatively, the learner might itself propose board states that it finds particularly confusing and ask the teacher for the correct move. Or the learner may have complete control over both the board states and (indirect) training classifications, as it does when it learns by playing against itself with no teacher

# CHAPTER 2

# CONCEPT LEARNING AND THE GENERAL-TO-SPECIFIC ORDERING

The problem of inducing general functions from specific training examples is central to learning. This chapter considers concept learning: acquiring the definition of a general category given a sample of positive and negative training examples of the category. Concept learning can be formulated as a problem of searching through a predefined space of potential hypotheses for the hypothesis that best fits the training examples. In many cases this search can be efficiently organized by taking advantage of a naturally occurring structure over the hypothesis space—a general-to-specific ordering of hypotheses. This chapter presents several learning algorithms and considers situations under which they converge to the correct hypothesis. We also examine the nature of inductive learning and the justification by which any program may successfully generalize beyond the observed training data.

## 2.1 INTRODUCTION

Much of learning involves acquiring general concepts from specific training examples. People, for example, continually learn general concepts or categories such as "bird," "car," "situations in which I should study more in order to pass the exam," etc. Each such concept can be viewed as describing some subset of objects or events defined over a larger set (e.g., the subset of animals that constitute

birds). Alternatively, each concept can be thought of as a boolean-valued function defined over this larger set (e.g., a function defined over all animals, whose value is true for birds and false for other animals).

In this chapter we consider the problem of automatically inferring the general definition of some concept, given examples labeled as members or nonmembers of the concept. This task is commonly referred to as *concept learning*, or approximating a boolean-valued function from examples.

**Concept learning.** Inferring a boolean-valued function from training examples of its input and output.

## 2.2 A CONCEPT LEARNING TASK

To ground our discussion of concept learning, consider the example task of learning the target concept "days on which my friend Aldo enjoys his favorite water sport." Table 2.1 describes a set of example days, each represented by a set of *attributes*. The attribute *EnjoySport* indicates whether or not Aldo enjoys his favorite water sport on this day. The task is to learn to predict the value of *EnjoySport* for an arbitrary day, based on the values of its other attributes.

What hypothesis representation shall we provide to the learner in this case? Let us begin by considering a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes. In particular, let each hypothesis be a vector of six constraints, specifying the values of the six attributes *Sky, AirTemp, Humidity, Wind, Water,* and *Forecast*. For each attribute, the hypothesis will either

- indicate by a "?" that any value is acceptable for this attribute,
- specify a single required value (e.g., *Warm*) for the attribute, or
- indicate by a "Ø" that no value is acceptable.

If some instance $x$ satisfies all the constraints of hypothesis $h$, then $h$ classifies $x$ as a positive example ($h(x) = 1$). To illustrate, the hypothesis that Aldo enjoys his favorite sport only on cold days with high humidity (independent of the values of the other attributes) is represented by the expression

$$\langle ?, Cold, High, ?, ?, ? \rangle$$

| Example | *Sky* | *AirTemp* | *Humidity* | *Wind* | *Water* | *Forecast* | *EnjoySport* |
|---|---|---|---|---|---|---|---|
| 1 | Sunny | Warm | Normal | Strong | Warm | Same | Yes |
| 2 | Sunny | Warm | High | Strong | Warm | Same | Yes |
| 3 | Rainy | Cold | High | Strong | Warm | Change | No |
| 4 | Sunny | Warm | High | Strong | Cool | Change | Yes |

**TABLE 2.1**
Positive and negative training examples for the target concept *EnjoySport*.

The most general hypothesis—that every day is a positive example—is represented by

$$\langle ?, ?, ?, ?, ?, ? \rangle$$

and the most specific possible hypothesis—that *no* day is a positive example—is represented by

$$\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

To summarize, the *EnjoySport* concept learning task requires learning the set of days for which *EnjoySport* = *yes*, describing this set by a conjunction of constraints over the instance attributes. In general, any concept learning task can be described by the set of instances over which the target function is defined, the target function, the set of candidate hypotheses considered by the learner, and the set of available training examples. The definition of the *EnjoySport* concept learning task in this general form is given in Table 2.2.

### 2.2.1 Notation

Throughout this book, we employ the following terminology when discussing concept learning problems. The set of items over which the concept is defined is called the set of *instances*, which we denote by $X$. In the current example, $X$ is the set of all possible days, each represented by the attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*. The concept or function to be learned is called the *target concept*, which we denote by $c$. In general, $c$ can be any boolean-valued function defined over the instances $X$; that is, $c : X \rightarrow \{0, 1\}$. In the current example, the target concept corresponds to the value of the attribute *EnjoySport* (i.e., $c(x) = 1$ if *EnjoySport* = *Yes*, and $c(x) = 0$ if *EnjoySport* = *No*).

- **Given:**
  - Instances $X$: Possible days, each described by the attributes
    - *Sky* (with possible values *Sunny*, *Cloudy*, and *Rainy*),
    - *AirTemp* (with values *Warm* and *Cold*),
    - *Humidity* (with values *Normal* and *High*),
    - *Wind* (with values *Strong* and *Weak*),
    - *Water* (with values *Warm* and *Cool*), and
    - *Forecast* (with values *Same* and *Change*).
  - Hypotheses $H$: Each hypothesis is described by a conjunction of constraints on the attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*. The constraints may be "?" (any value is acceptable), "$\emptyset$" (no value is acceptable), or a specific value.
  - Target concept $c$: *EnjoySport* : $X \rightarrow \{0, 1\}$
  - Training examples $D$: Positive and negative examples of the target function (see Table 2.1).
- **Determine:**
  - A hypothesis $h$ in $H$ such that $h(x) = c(x)$ for all $x$ in $X$.

**TABLE 2.2**
The *EnjoySport* concept learning task.

When learning the target concept, the learner is presented a set of *training examples*, each consisting of an instance $x$ from $X$, along with its target concept value $c(x)$ (e.g., the training examples in Table 2.1). Instances for which $c(x) = 1$ are called *positive examples*, or members of the target concept. Instances for which $c(x) = 0$ are called *negative examples*, or nonmembers of the target concept. We will often write the ordered pair $\langle x, c(x) \rangle$ to describe the training example consisting of the instance $x$ and its target concept value $c(x)$. We use the symbol $D$ to denote the set of available training examples.

Given a set of training examples of the target concept $c$, the problem faced by the learner is to hypothesize, or estimate, $c$. We use the symbol $H$ to denote the set of *all possible hypotheses* that the learner may consider regarding the identity of the target concept. Usually $H$ is determined by the human designer's choice of hypothesis representation. In general, each hypothesis $h$ in $H$ represents a boolean-valued function defined over $X$; that is, $h : X \rightarrow \{0, 1\}$. The goal of the learner is to find a hypothesis $h$ such that $h(x) = c(x)$ for all $x$ in $X$.

### 2.2.2 The Inductive Learning Hypothesis

Notice that although the learning task is to determine a hypothesis $h$ identical to the target concept $c$ over the entire set of instances $X$, the only information available about $c$ is its value over the training examples. Therefore, inductive learning algorithms can at best guarantee that the output hypothesis fits the target concept over the training data. Lacking any further information, our assumption is that the best hypothesis regarding unseen instances is the hypothesis that best fits the observed training data. This is the fundamental assumption of inductive learning, and we will have much more to say about it throughout this book. We state it here informally and will revisit and analyze this assumption more formally and more quantitatively in Chapters 5, 6, and 7.

> **The inductive learning hypothesis.** Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

## 2.3 CONCEPT LEARNING AS SEARCH

Concept learning can be viewed as the task of searching through a large space of hypotheses implicitly defined by the hypothesis representation. The goal of this search is to find the hypothesis that best fits the training examples. It is important to note that by selecting a hypothesis representation, the designer of the learning algorithm implicitly defines the space of all hypotheses that the program can ever represent and therefore can ever learn. Consider, for example, the instances $X$ and hypotheses $H$ in the *EnjoySport* learning task. Given that the attribute *Sky* has three possible values, and that *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast* each have two possible values, the instance space $X$ contains exactly

$3 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 96$ distinct instances. A similar calculation shows that there are $5 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 = 5120$ *syntactically distinct* hypotheses within $H$. Notice, however, that every hypothesis containing one or more "$\emptyset$" symbols represents the empty set of instances; that is, it classifies every instance as negative. Therefore, the number of *semantically distinct* hypotheses is only $1 + (4 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3) = 973$. Our *EnjoySport* example is a very simple learning task, with a relatively small, finite hypothesis space. Most practical learning tasks involve much larger, sometimes infinite, hypothesis spaces.

If we view learning as a search problem, then it is natural that our study of learning algorithms will examine different strategies for searching the hypothesis space. We will be particularly interested in algorithms capable of efficiently searching very large or infinite hypothesis spaces, to find the hypotheses that best fit the training data.

### 2.3.1 General-to-Specific Ordering of Hypotheses

Many algorithms for concept learning organize the search through the hypothesis space by relying on a very useful structure that exists for any concept learning problem: a general-to-specific ordering of hypotheses. By taking advantage of this naturally occurring structure over the hypothesis space, we can design learning algorithms that exhaustively search even infinite hypothesis spaces without explicitly enumerating every hypothesis. To illustrate the general-to-specific ordering, consider the two hypotheses

$$h_1 = \langle Sunny, ?, ?, Strong, ?, ? \rangle$$

$$h_2 = \langle Sunny, ?, ?, ?, ?, ? \rangle$$

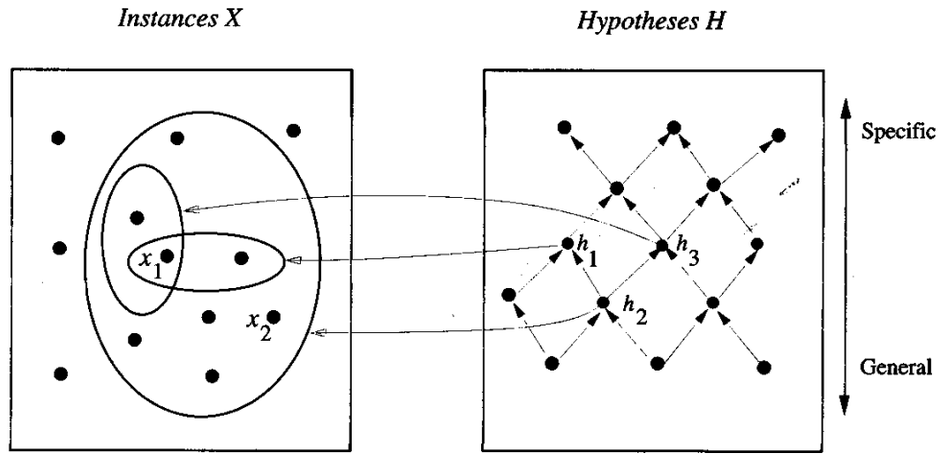
Now consider the sets of instances that are classified positive by $h_1$ and by $h_2$. Because $h_2$ imposes fewer constraints on the instance, it classifies more instances as positive. In fact, any instance classified positive by $h_1$ will also be classified positive by $h_2$. Therefore, we say that $h_2$ is more general than $h_1$.

This intuitive "more general than" relationship between hypotheses can be defined more precisely as follows. First, for any instance $x$ in $X$ and hypothesis $h$ in $H$, we say that $x$ *satisfies* $h$ if and only if $h(x) = 1$. We now define the *more_general_than_or_equal_to* relation in terms of the sets of instances that satisfy the two hypotheses: Given hypotheses $h_j$ and $h_k$, $h_j$ is *more_general_than_or_equal_to* $h_k$ if and only if any instance that satisfies $h_k$ also satisfies $h_j$.

***Definition***: Let $h_j$ and $h_k$ be boolean-valued functions defined over $X$. Then $h_j$ is **more_general_than_or_equal_to** $h_k$ (written $h_j \geq_g h_k$) if and only if

$$(\forall x \in X)[(h_k(x) = 1) \rightarrow (h_j(x) = 1)]$$

We will also find it useful to consider cases where one hypothesis is strictly more general than the other. Therefore, we will say that $h_j$ is (strictly) *more_general_than*




$x_1$= <Sunny, Warm, High, Strong, Cool, Same>
$x_2$= <Sunny, Warm, High, Light, Warm, Same>

$h_1$= <Sunny, ?, ?, Strong, ?, ?>
$h_2$= <Sunny, ?, ?, ?, ?, ?>
$h_3$= <Sunny, ?, ?, ?, Cool, ?>

**FIGURE 2.1**
Instances, hypotheses, and the *more_general_than* relation. The box on the left represents the set $X$ of all instances, the box on the right the set $H$ of all hypotheses. Each hypothesis corresponds to some subset of $X$—the subset of instances that it classifies positive. The arrows connecting hypotheses represent the *more_general_than* relation, with the arrow pointing toward the less general hypothesis. Note the subset of instances characterized by $h_2$ subsumes the subset characterized by $h_1$, hence $h_2$ is *more_general_than* $h_1$.

$h_k$ (written $h_j >_g h_k$) if and only if $(h_j \geq_g h_k) \wedge (h_k \not\geq_g h_j)$. Finally, we will sometimes find the inverse useful and will say that $h_j$ is *more_specific_than* $h_k$ when $h_k$ is *more_general_than* $h_j$.

To illustrate these definitions, consider the three hypotheses $h_1$, $h_2$, and $h_3$ from our *EnjoySport* example, shown in Figure 2.1. How are these three hypotheses related by the $\geq_g$ relation? As noted earlier, hypothesis $h_2$ is more general than $h_1$ because every instance that satisfies $h_1$ also satisfies $h_2$. Similarly, $h_2$ is more general than $h_3$. Note that neither $h_1$ nor $h_3$ is more general than the other; although the instances satisfied by these two hypotheses intersect, neither set subsumes the other. Notice also that the $\geq_g$ and $>_g$ relations are defined independent of the target concept. They depend only on which instances satisfy the two hypotheses and not on the classification of those instances according to the target concept. Formally, the $\geq_g$ relation defines a partial order over the hypothesis space $H$ (the relation is reflexive, antisymmetric, and transitive). Informally, when we say the structure is a partial (as opposed to total) order, we mean there may be pairs of hypotheses such as $h_1$ and $h_3$, such that $h_1 \not\geq_g h_3$ and $h_3 \not\geq_g h_1$.

The $\geq_g$ relation is important because it provides a useful structure over the hypothesis space $H$ for *any* concept learning problem. The following sections present concept learning algorithms that take advantage of this partial order to efficiently organize the search for hypotheses that fit the training data.

1. Initialize $h$ to the most specific hypothesis in $H$
2. For each positive training instance $x$
   - For each attribute constraint $a_i$ in $h$
     If the constraint $a_i$ is satisfied by $x$
     Then do nothing
     Else replace $a_i$ in $h$ by the next more general constraint that is satisfied by $x$
3. Output hypothesis $h$

**TABLE 2.3**
FIND-S Algorithm.

## 2.4 FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS

How can we use the *more_general_than* partial ordering to organize the search for a hypothesis consistent with the observed training examples? One way is to begin with the most specific possible hypothesis in $H$, then generalize this hypothesis each time it fails to cover an observed positive training example. (We say that a hypothesis "covers" a positive example if it correctly classifies the example as positive.) To be more precise about how the partial ordering is used, consider the FIND-S algorithm defined in Table 2.3.

To illustrate this algorithm, assume the learner is given the sequence of training examples from Table 2.1 for the *EnjoySport* task. The first step of FIND-S is to initialize $h$ to the most specific hypothesis in $H$

$$h \leftarrow \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

Upon observing the first training example from Table 2.1, which happens to be a positive example, it becomes clear that our hypothesis is too specific. In particular, none of the "$\emptyset$" constraints in $h$ are satisfied by this example, so each is replaced by the next more general constraint that fits the example; namely, the attribute values for this training example.

$$h \leftarrow \langle Sunny, Warm, Normal, Strong, Warm, Same \rangle$$

This $h$ is still very specific; it asserts that all instances are negative except for the single positive training example we have observed. Next, the second training example (also positive in this case) forces the algorithm to further generalize $h$, this time substituting a "?" in place of any attribute value in $h$ that is not satisfied by the new example. The refined hypothesis in this case is

$$h \leftarrow \langle Sunny, Warm, ?, Strong, Warm, Same \rangle$$

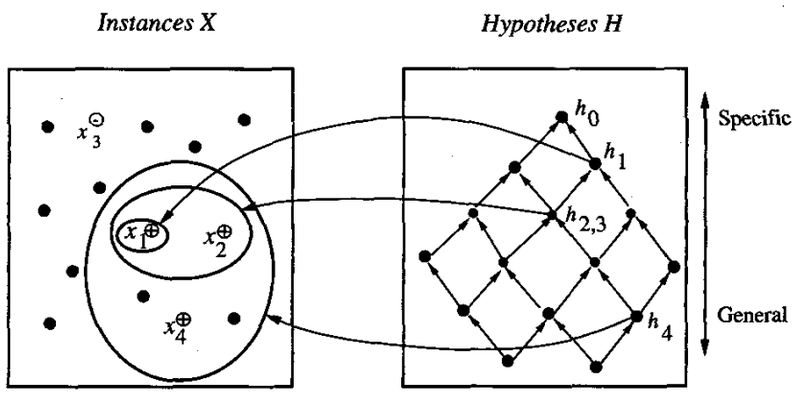
Upon encountering the third training example—in this case a negative example—the algorithm makes no change to $h$. In fact, the FIND-S algorithm simply *ignores every negative example*! While this may at first seem strange, notice that in the current case our hypothesis $h$ is already consistent with the new negative example (i.e., $h$ correctly classifies this example as negative), and hence no revision

is needed. In the general case, as long as we assume that the hypothesis space $H$ contains a hypothesis that describes the true target concept $c$ and that the training data contains no errors, then the current hypothesis $h$ can never require a revision in response to a negative example. To see why, recall that the current hypothesis $h$ is the most specific hypothesis in $H$ consistent with the observed positive examples. Because the target concept $c$ is also assumed to be in $H$ and to be consistent with the positive training examples, $c$ must be *more_general_than_or_equal_to* $h$. But the target concept $c$ will never cover a negative example, thus neither will $h$ (by the definition of *more_general_than*). Therefore, no revision to $h$ will be required in response to any negative example.

To complete our trace of FIND-S, the fourth (positive) example leads to a further generalization of $h$

$$h \leftarrow \langle Sunny, Warm, ?, Strong, ?, ? \rangle$$

The FIND-S algorithm illustrates one way in which the *more_general_than* partial ordering can be used to organize the search for an acceptable hypothesis. The search moves from hypothesis to hypothesis, searching from the most specific to progressively more general hypotheses along one chain of the partial ordering. Figure 2.2 illustrates this search in terms of the instance and hypothesis spaces. At each step, the hypothesis is generalized only as far as necessary to cover the new positive example. Therefore, at each stage the hypothesis is the most specific hypothesis consistent with the training examples observed up to this point (hence the name FIND-S). The literature on concept learning is




$x_1$ = <*Sunny Warm Normal Strong Warm Same*>, +
$x_2$ = <*Sunny Warm High Strong Warm Same*>, +
$x_3$ = <*Rainy Cold High Strong Warm Change*>, -
$x_4$ = <*Sunny Warm High Strong Cool Change*>, +

$h_0$ = <∅, ∅, ∅, ∅, ∅, ∅>
$h_1$ = <*Sunny Warm Normal Strong Warm Same*>
$h_2$ = <*Sunny Warm ? Strong Warm Same*>
$h_3$ = <*Sunny Warm ? Strong Warm Same*>
$h_4$ = <*Sunny Warm ? Strong ? ?*>

**FIGURE 2.2**
The hypothesis space search performed by FIND-S. The search begins ($h_0$) with the most specific hypothesis in $H$, then considers increasingly general hypotheses ($h_1$ through $h_4$) as mandated by the training examples. In the instance space diagram, positive training examples are denoted by "+," negative by "–," and instances that have not been presented as training examples are denoted by a solid circle.

populated by many different algorithms that utilize this same *more_general_than* partial ordering to organize the search in one fashion or another. A number of such algorithms are discussed in this chapter, and several others are presented in Chapter 10.

The key property of the FIND-S algorithm is that for hypothesis spaces described by conjunctions of attribute constraints (such as $H$ for the *EnjoySport* task), FIND-S is guaranteed to output the most specific hypothesis within $H$ that is consistent with the positive training examples. Its final hypothesis will also be consistent with the negative examples provided the correct target concept is contained in $H$, and provided the training examples are correct. However, there are several questions still left unanswered by this learning algorithm, such as:

- Has the learner converged to the correct target concept? Although FIND-S will find a hypothesis consistent with the training data, it has no way to determine whether it has found the *only* hypothesis in $H$ consistent with the data (i.e., the correct target concept), or whether there are many other consistent hypotheses as well. We would prefer a learning algorithm that could determine whether it had converged and, if not, at least characterize its uncertainty regarding the true identity of the target concept.

- Why prefer the most specific hypothesis? In case there are multiple hypotheses consistent with the training examples, FIND-S will find the most specific. It is unclear whether we should prefer this hypothesis over, say, the most general, or some other hypothesis of intermediate generality.

- Are the training examples consistent? In most practical learning problems there is some chance that the training examples will contain at least some errors or noise. Such inconsistent sets of training examples can severely mislead FIND-S, given the fact that it ignores negative examples. We would prefer an algorithm that could at least detect when the training data is inconsistent and, preferably, accommodate such errors.

- What if there are several maximally specific consistent hypotheses? In the hypothesis language $H$ for the *EnjoySport* task, there is always a unique, most specific hypothesis consistent with any set of positive examples. However, for other hypothesis spaces (discussed later) there can be several maximally specific hypotheses consistent with the data. In this case, FIND-S must be extended to allow it to backtrack on its choices of how to generalize the hypothesis, to accommodate the possibility that the target concept lies along a different branch of the partial ordering than the branch it has selected. Furthermore, we can define hypothesis spaces for which there is no maximally specific consistent hypothesis, although this is more of a theoretical issue than a practical one (see Exercise 2.7).

# CHAPTER 6

# BAYESIAN LEARNING

Bayesian reasoning provides a probabilistic approach to inference. It is based on the assumption that the quantities of interest are governed by probability distributions and that optimal decisions can be made by reasoning about these probabilities together with observed data. It is important to machine learning because it provides a quantitative approach to weighing the evidence supporting alternative hypotheses. Bayesian reasoning provides the basis for learning algorithms that directly manipulate probabilities, as well as a framework for analyzing the operation of other algorithms that do not explicitly manipulate probabilities.

## 6.1 INTRODUCTION

Bayesian learning methods are relevant to our study of machine learning for two different reasons. First, Bayesian learning algorithms that calculate explicit probabilities for hypotheses, such as the naive Bayes classifier, are among the most practical approaches to certain types of learning problems. For example, Michie et al. (1994) provide a detailed study comparing the naive Bayes classifier to other learning algorithms, including decision tree and neural network algorithms. These researchers show that the naive Bayes classifier is competitive with these other learning algorithms in many cases and that in some cases it outperforms these other methods. In this chapter we describe the naive Bayes classifier and provide a detailed example of its use. In particular, we discuss its application to the problem of learning to classify text documents such as electronic news articles.

For such learning tasks, the naive Bayes classifier is among the most effective algorithms known.

The second reason that Bayesian methods are important to our study of machine learning is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities. For example, in this chapter we analyze algorithms such as the FIND-S and CANDIDATE-ELIMINATION algorithms of Chapter 2 to determine conditions under which they output the most probable hypothesis given the training data. We also use a Bayesian analysis to justify a key design choice in neural network learning algorithms: choosing to minimize the sum of squared errors when searching the space of possible neural networks. We also derive an alternative error function, cross entropy, that is more appropriate than sum of squared errors when learning target functions that predict probabilities. We use a Bayesian perspective to analyze the inductive bias of decision tree learning algorithms that favor short decision trees and examine the closely related Minimum Description Length principle. A basic familiarity with Bayesian methods is important to understanding and characterizing the operation of many algorithms in machine learning.

Features of Bayesian learning methods include:

- Each observed training example can incrementally decrease or increase the estimated probability that a hypothesis is correct. This provides a more flexible approach to learning than algorithms that completely eliminate a hypothesis if it is found to be inconsistent with any single example.
- Prior knowledge can be combined with observed data to determine the final probability of a hypothesis. In Bayesian learning, prior knowledge is provided by asserting (1) a prior probability for each candidate hypothesis, and (2) a probability distribution over observed data for each possible hypothesis.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions (e.g., hypotheses such as "this pneumonia patient has a 93% chance of complete recovery").
- New instances can be classified by combining the predictions of multiple hypotheses, weighted by their probabilities.
- Even in cases where Bayesian methods prove computationally intractable, they can provide a standard of optimal decision making against which other practical methods can be measured.

One practical difficulty in applying Bayesian methods is that they typically require initial knowledge of many probabilities. When these probabilities are not known in advance they are often estimated based on background knowledge, previously available data, and assumptions about the form of the underlying distributions. A second practical difficulty is the significant computational cost required to determine the Bayes optimal hypothesis in the general case (linear in the number of candidate hypotheses). In certain specialized situations, this computational cost can be significantly reduced.

The remainder of this chapter is organized as follows. Section 6.2 introduces Bayes theorem and defines maximum likelihood and maximum a posteriori probability hypotheses. The four subsequent sections then apply this probabilistic framework to analyze several issues and learning algorithms discussed in earlier chapters. For example, we show that several previously described algorithms output maximum likelihood hypotheses, under certain assumptions. The remaining sections then introduce a number of learning algorithms that explicitly manipulate probabilities. These include the Bayes optimal classifier, Gibbs algorithm, and naive Bayes classifier. Finally, we discuss Bayesian belief networks, a relatively recent approach to learning based on probabilistic reasoning, and the EM algorithm, a widely used algorithm for learning in the presence of unobserved variables.

## 6.2 BAYES THEOREM

In machine learning we are often interested in determining the best hypothesis from some space $H$, given the observed training data $D$. One way to specify what we mean by the *best* hypothesis is to say that we demand the *most probable* hypothesis, given the data $D$ plus any initial knowledge about the prior probabilities of the various hypotheses in $H$. Bayes theorem provides a direct method for calculating such probabilities. More precisely, Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.

To define Bayes theorem precisely, let us first introduce a little notation. We shall write $P(h)$ to denote the initial probability that hypothesis $h$ holds, before we have observed the training data. $P(h)$ is often called the *prior probability* of $h$ and may reflect any background knowledge we have about the chance that $h$ is a correct hypothesis. If we have no such prior knowledge, then we might simply assign the same prior probability to each candidate hypothesis. Similarly, we will write $P(D)$ to denote the prior probability that training data $D$ will be observed (i.e., the probability of $D$ given no knowledge about which hypothesis holds). Next, we will write $P(D|h)$ to denote the probability of observing data $D$ given some world in which hypothesis $h$ holds. More generally, we write $P(x|y)$ to denote the probability of $x$ given $y$. In machine learning problems we are interested in the probability $P(h|D)$ that $h$ holds given the observed training data $D$. $P(h|D)$ is called the *posterior probability* of $h$, because it reflects our confidence that $h$ holds after we have seen the training data $D$. Notice the posterior probability $P(h|D)$ reflects the influence of the training data $D$, in contrast to the prior probability $P(h)$, which is independent of $D$.

Bayes theorem is the cornerstone of Bayesian learning methods because it provides a way to calculate the posterior probability $P(h|D)$, from the prior probability $P(h)$, together with $P(D)$ and $P(D|h)$.

**Bayes theorem:**

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \tag{6.1}$$

As one might intuitively expect, $P(h|D)$ increases with $P(h)$ and with $P(D|h)$ according to Bayes theorem. It is also reasonable to see that $P(h|D)$ decreases as $P(D)$ increases, because the more probable it is that $D$ will be observed independent of $h$, the less evidence $D$ provides in support of $h$.

In many learning scenarios, the learner considers some set of candidate hypotheses $H$ and is interested in finding the most probable hypothesis $h \in H$ given the observed data $D$ (or at least one of the maximally probable if there are several). Any such maximally probable hypothesis is called a *maximum a posteriori* (MAP) hypothesis. We can determine the MAP hypotheses by using Bayes theorem to calculate the posterior probability of each candidate hypothesis. More precisely, we will say that $h_{MAP}$ is a MAP hypothesis provided

$$\begin{aligned} h_{MAP} &\equiv \underset{h \in H}{\operatorname{argmax}} P(h|D) \\ &= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)} \\ &= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \end{aligned} \tag{6.2}$$

Notice in the final step above we dropped the term $P(D)$ because it is a constant independent of $h$.

In some cases, we will assume that every hypothesis in $H$ is equally probable a priori ($P(h_i) = P(h_j)$ for all $h_i$ and $h_j$ in $H$). In this case we can further simplify Equation (6.2) and need only consider the term $P(D|h)$ to find the most probable hypothesis. $P(D|h)$ is often called the *likelihood* of the data $D$ given $h$, and any hypothesis that maximizes $P(D|h)$ is called a *maximum likelihood* (ML) hypothesis, $h_{ML}$.

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D|h) \tag{6.3}$$

In order to make clear the connection to machine learning problems, we introduced Bayes theorem above by referring to the data $D$ as training examples of some target function and referring to $H$ as the space of candidate target functions. In fact, Bayes theorem is much more general than suggested by this discussion. It can be applied equally well to any set $H$ of mutually exclusive propositions whose probabilities sum to one (e.g., "the sky is blue," and "the sky is not blue"). In this chapter, we will at times consider cases where $H$ is a hypothesis space containing possible target functions and the data $D$ are training examples. At other times we will consider cases where $H$ is some other set of mutually exclusive propositions, and $D$ is some other kind of data.

### 6.2.1 An Example

To illustrate Bayes rule, consider a medical diagnosis problem in which there are two alternative hypotheses: (1) that the patient has a particular form of cancer, and (2) that the patient does not. The available data is from a particular laboratory

test with two possible outcomes: $\oplus$ (positive) and $\ominus$ (negative). We have prior knowledge that over the entire population of people only .008 have this disease. Furthermore, the lab test is only an imperfect indicator of the disease. The test returns a correct positive result in only 98% of the cases in which the disease is actually present and a correct negative result in only 97% of the cases in which the disease is not present. In other cases, the test returns the opposite result. The above situation can be summarized by the following probabilities:

$$\begin{aligned} P(cancer) &= .008, & P(\neg cancer) &= .992 \\ P(\oplus|cancer) &= .98, & P(\ominus|cancer) &= .02 \\ P(\oplus|\neg cancer) &= .03, & P(\ominus|\neg cancer) &= .97 \end{aligned}$$

Suppose we now observe a new patient for whom the lab test returns a positive result. Should we diagnose the patient as having cancer or not? The maximum a posteriori hypothesis can be found using Equation (6.2):

$$P(\oplus|cancer)P(cancer) = (.98).008 = .0078$$

$$P(\oplus|\neg cancer)P(\neg cancer) = (.03).992 = .0298$$

Thus, $h_{MAP} = \neg cancer$. The exact posterior probabilities can also be determined by normalizing the above quantities so that they sum to 1 (e.g., $P(cancer|\oplus) = \frac{.0078}{.0078+.0298} = .21$). This step is warranted because Bayes theorem states that the posterior probabilities are just the above quantities divided by the probability of the data, $P(\oplus)$. Although $P(\oplus)$ was not provided directly as part of the problem statement, we can calculate it in this fashion because we know that $P(cancer|\oplus)$ and $P(\neg cancer|\oplus)$ must sum to 1 (i.e., either the patient has cancer or they do not). Notice that while the posterior probability of *cancer* is significantly higher than its prior probability, the most probable hypothesis is still that the patient does not have cancer.

As this example illustrates, the result of Bayesian inference depends strongly on the prior probabilities, which must be available in order to apply the method directly. Note also that in this example the hypotheses are not completely accepted or rejected, but rather become more or less probable as more data is observed.

Basic formulas for calculating probabilities are summarized in Table 6.1.

## 6.3 BAYES THEOREM AND CONCEPT LEARNING

What is the relationship between Bayes theorem and the problem of concept learning? Since Bayes theorem provides a principled way to calculate the posterior probability of each hypothesis given the training data, we can use it as the basis for a straightforward learning algorithm that calculates the probability for each possible hypothesis, then outputs the most probable. This section considers such a brute-force Bayesian concept learning algorithm, then compares it to concept learning algorithms we considered in Chapter 2. As we shall see, one interesting result of this comparison is that under certain conditions several algorithms discussed in earlier chapters output the same hypotheses as this brute-force Bayesian

as its correct classification (which can be done using at most $\log_2 k$ bits, where $k$ is the number of possible classifications). The hypothesis $h_{MDL}$ under the encodings $C_1$ and $C_2$ is just the one that minimizes the sum of these description lengths.

Thus the MDL principle provides a way of trading off hypothesis complexity for the number of errors committed by the hypothesis. It might select a shorter hypothesis that makes a few errors over a longer hypothesis that perfectly classifies the training data. Viewed in this light, it provides one method for dealing with the issue of *overfitting* the data.

Quinlan and Rivest (1989) describe experiments applying the MDL principle to choose the best size for a decision tree. They report that the MDL-based method produced learned trees whose accuracy was comparable to that of the standard tree-pruning methods discussed in Chapter 3. Mehta et al. (1995) describe an alternative MDL-based approach to decision tree pruning, and describe experiments in which an MDL-based approach produced results comparable to standard tree-pruning methods.

What shall we conclude from this analysis of the Minimum Description Length principle? Does this prove once and for all that short hypotheses are best? No. What we have shown is only that *if* a representation of hypotheses is chosen so that the size of hypothesis $h$ is $-\log_2 P(h)$, and *if* a representation for exceptions is chosen so that the encoding length of $D$ given $h$ is equal to $-\log_2 P(D|h)$, *then* the MDL principle produces MAP hypotheses. However, to show that we have such a representation we must know all the prior probabilities $P(h)$, as well as the $P(D|h)$. There is no reason to believe that the MDL hypothesis relative to *arbitrary* encodings $C_1$ and $C_2$ should be preferred. As a practical matter it might sometimes be easier for a human designer to specify a representation that captures knowledge about the relative probabilities of hypotheses than it is to fully specify the probability of each hypothesis. Descriptions in the literature on the application of MDL to practical learning problems often include arguments providing some form of justification for the encodings chosen for $C_1$ and $C_2$.

## 6.7 BAYES OPTIMAL CLASSIFIER

So far we have considered the question "what is the most probable *hypothesis* given the training data?" In fact, the question that is often of most significance is the closely related question "what is the most probable *classification* of the new instance given the training data?" Although it may seem that this second question can be answered by simply applying the MAP hypothesis to the new instance, in fact it is possible to do better.

To develop some intuitions consider a hypothesis space containing three hypotheses, $h_1$, $h_2$, and $h_3$. Suppose that the posterior probabilities of these hypotheses given the training data are .4, .3, and .3 respectively. Thus, $h_1$ is the MAP hypothesis. Suppose a new instance $x$ is encountered, which is classified positive by $h_1$, but negative by $h_2$ and $h_3$. Taking all hypotheses into account, the probability that $x$ is positive is .4 (the probability associated with $h_1$), and

the probability that it is negative is therefore .6. The most probable classification (negative) in this case is different from the classification generated by the MAP hypothesis.

In general, the most probable classification of the new instance is obtained by combining the predictions of all hypotheses, weighted by their posterior probabilities. If the possible classification of the new example can take on any value $v_j$ from some set $V$, then the probability $P(v_j|D)$ that the correct classification for the new instance is $v_j$, is just

$$P(v_j|D) = \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

The optimal classification of the new instance is the value $v_j$, for which $P(v_j|D)$ is maximum.

**Bayes optimal classification:**

$$\underset{v_j \in V}{\operatorname{argmax}} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D) \tag{6.18}$$

To illustrate in terms of the above example, the set of possible classifications of the new instance is $V = \{\oplus, \ominus\}$, and

$$P(h_1|D) = .4, \; P(\ominus|h_1) = 0, \; P(\oplus|h_1) = 1$$
$$P(h_2|D) = .3, \; P(\ominus|h_2) = 1, \; P(\oplus|h_2) = 0$$
$$P(h_3|D) = .3, \; P(\ominus|h_3) = 1, \; P(\oplus|h_3) = 0$$

therefore

$$\sum_{h_i \in H} P(\oplus|h_i)P(h_i|D) = .4$$
$$\sum_{h_i \in H} P(\ominus|h_i)P(h_i|D) = .6$$

and

$$\underset{v_j \in \{\oplus,\ominus\}}{\operatorname{argmax}} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D) = \ominus$$

Any system that classifies new instances according to Equation (6.18) is called a *Bayes optimal classifier*, or Bayes optimal learner. No other classification method using the same hypothesis space and same prior knowledge can outperform this method on average. This method maximizes the probability that the new instance is classified correctly, given the available data, hypothesis space, and prior probabilities over the hypotheses.

For example, in learning boolean concepts using version spaces as in the earlier section, the Bayes optimal classification of a new instance is obtained by taking a weighted vote among all members of the version space, with each candidate hypothesis weighted by its posterior probability.

Note one curious property of the Bayes optimal classifier is that the predictions it makes can correspond to a hypothesis not contained in $H$! Imagine using Equation (6.18) to classify every instance in $X$. The labeling of instances defined in this way need not correspond to the instance labeling of any single hypothesis $h$ from $H$. One way to view this situation is to think of the Bayes optimal classifier as effectively considering a hypothesis space $H'$ different from the space of hypotheses $H$ to which Bayes theorem is being applied. In particular, $H'$ effectively includes hypotheses that perform comparisons between linear combinations of predictions from multiple hypotheses in $H$.

## 6.8 GIBBS ALGORITHM

Although the Bayes optimal classifier obtains the best performance that can be achieved from the given training data, it can be quite costly to apply. The expense is due to the fact that it computes the posterior probability for every hypothesis in $H$ and then combines the predictions of each hypothesis to classify each new instance.

An alternative, less optimal method is the Gibbs algorithm (see Opper and Haussler 1991), defined as follows:

1. Choose a hypothesis $h$ from $H$ at random, according to the posterior probability distribution over $H$.
2. Use $h$ to predict the classification of the next instance $x$.

Given a new instance to classify, the Gibbs algorithm simply applies a hypothesis drawn at random according to the current posterior probability distribution. Surprisingly, it can be shown that under certain conditions the expected misclassification error for the Gibbs algorithm is at most twice the expected error of the Bayes optimal classifier (Haussler et al. 1994). More precisely, the expected value is taken over target concepts drawn at random according to the prior probability distribution assumed by the learner. Under this condition, the expected value of the error of the Gibbs algorithm is at worst twice the expected value of the error of the Bayes optimal classifier.

This result has an interesting implication for the concept learning problem described earlier. In particular, it implies that if the learner assumes a uniform prior over $H$, and if target concepts are in fact drawn from such a distribution when presented to the learner, *then classifying the next instance according to a hypothesis drawn at random from the current version space (according to a uniform distribution), will have expected error at most twice that of the Bayes optimal classifier.* Again, we have an example where a Bayesian analysis of a non-Bayesian algorithm yields insight into the performance of that algorithm.

## 6.9 NAIVE BAYES CLASSIFIER

One highly practical Bayesian learning method is the naive Bayes learner, often called the *naive Bayes classifier*. In some domains its performance has been shown to be comparable to that of neural network and decision tree learning. This section introduces the naive Bayes classifier; the next section applies it to the practical problem of learning to classify natural language text documents.

The naive Bayes classifier applies to learning tasks where each instance $x$ is described by a conjunction of attribute values and where the target function $f(x)$ can take on any value from some finite set $V$. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values $\langle a_1, a_2 \ldots a_n \rangle$. The learner is asked to predict the target value, or classification, for this new instance.

The Bayesian approach to classifying the new instance is to assign the most probable target value, $v_{MAP}$, given the attribute values $\langle a_1, a_2 \ldots a_n \rangle$ that describe the instance.

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j | a_1, a_2 \ldots a_n)$$

We can use Bayes theorem to rewrite this expression as

$$\begin{aligned} v_{MAP} &= \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)} \\ &= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j) \end{aligned} \tag{6.19}$$

Now we could attempt to estimate the two terms in Equation (6.19) based on the training data. It is easy to estimate each of the $P(v_j)$ simply by counting the frequency with which each target value $v_j$ occurs in the training data. However, estimating the different $P(a_1, a_2 \ldots a_n | v_j)$ terms in this fashion is not feasible unless we have a very, very large set of training data. The problem is that the number of these terms is equal to the number of possible instances times the number of possible target values. Therefore, we need to see every instance in the instance space many times in order to obtain reliable estimates.

The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value. In other words, the assumption is that given the target value of the instance, the probability of observing the conjunction $a_1, a_2 \ldots a_n$ is just the product of the probabilities for the individual attributes: $P(a_1, a_2 \ldots a_n | v_j) = \prod_i P(a_i | v_j)$. Substituting this into Equation (6.19), we have the approach used by the naive Bayes classifier.

**Naive Bayes classifier:**

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i | v_j) \tag{6.20}$$

where $v_{NB}$ denotes the target value output by the naive Bayes classifier. Notice that in a naive Bayes classifier the number of distinct $P(a_i | v_j)$ terms that must

be estimated from the training data is just the number of distinct attribute values times the number of distinct target values—a much smaller number than if we were to estimate the $P(a_1, a_2 \ldots a_n | v_j)$ terms as first contemplated.

To summarize, the naive Bayes learning method involves a learning step in which the various $P(v_j)$ and $P(a_i|v_j)$ terms are estimated, based on their frequencies over the training data. The set of these estimates corresponds to the learned hypothesis. This hypothesis is then used to classify each new instance by applying the rule in Equation (6.20). Whenever the naive Bayes assumption of conditional independence is satisfied, this naive Bayes classification $v_{NB}$ is identical to the MAP classification.

One interesting difference between the naive Bayes learning method and other learning methods we have considered is that there is no explicit search through the space of possible hypotheses (in this case, the space of possible hypotheses is the space of possible values that can be assigned to the various $P(v_j)$ and $P(a_i|v_j)$ terms). Instead, the hypothesis is formed without searching, simply by counting the frequency of various data combinations within the training examples.

### 6.9.1 An Illustrative Example

Let us apply the naive Bayes classifier to a concept learning problem we considered during our discussion of decision tree learning: classifying days according to whether someone will play tennis. Table 3.2 from Chapter 3 provides a set of 14 training examples of the target concept *PlayTennis*, where each day is described by the attributes *Outlook, Temperature, Humidity*, and *Wind*. Here we use the naive Bayes classifier and the training data from this table to classify the following novel instance:

$$\langle Outlook = sunny, Temperature = cool, Humidity = high, Wind = strong \rangle$$

Our task is to predict the target value (*yes* or *no*) of the target concept *PlayTennis* for this new instance. Instantiating Equation (6.20) to fit the current task, the target value $v_{NB}$ is given by

$$\begin{aligned} v_{NB} &= \underset{v_j \in \{yes,no\}}{\operatorname{argmax}} P(v_j) \textstyle\prod_i P(a_i|v_j) \\ &= \underset{v_j \in \{yes,no\}}{\operatorname{argmax}} P(v_j) \quad P(Outlook = sunny|v_j) P(Temperature = cool|v_j) \\ &\qquad \cdot P(Humidity = high|v_j) P(Wind = strong|v_j) \end{aligned} \tag{6.21}$$

Notice in the final expression that $a_i$ has been instantiated using the particular attribute values of the new instance. To calculate $v_{NB}$ we now require 10 probabilities that can be estimated from the training data. First, the probabilities of the different target values can easily be estimated based on their frequencies over the 14 training examples

$$P(PlayTennis = yes) = 9/14 = .64$$

$$P(PlayTennis = no) = 5/14 = .36$$

Similarly, we can estimate the conditional probabilities. For example, those for $Wind = strong$ are

$$P(Wind = strong | PlayTennis = yes) = 3/9 = .33$$

$$P(Wind = strong | PlayTennis = no) = 3/5 = .60$$

Using these probability estimates and similar estimates for the remaining attribute values, we calculate $v_{NB}$ according to Equation (6.21) as follows (now omitting attribute names for brevity)

$$P(yes)\ P(sunny|yes)\ P(cool|yes)\ P(high|yes)\ P(strong|yes) = .0053$$

$$P(no)\ P(sunny|no)\ P(cool|no)\ P(high|no)\ P(strong|no) = .0206$$

Thus, the naive Bayes classifier assigns the target value $PlayTennis = no$ to this new instance, based on the probability estimates learned from the training data. Furthermore, by normalizing the above quantities to sum to one we can calculate the conditional probability that the target value is *no*, given the observed attribute values. For the current example, this probability is $\frac{.0206}{.0206+.0053} = .795$.

#### 6.9.1.1 ESTIMATING PROBABILITIES

Up to this point we have estimated probabilities by the fraction of times the event is observed to occur over the total number of opportunities. For example, in the above case we estimated $P(Wind = strong | PlayTennis = no)$ by the fraction $\frac{n_c}{n}$ where $n = 5$ is the total number of training examples for which $PlayTennis = no$, and $n_c = 3$ is the number of these for which $Wind = strong$.

While this observed fraction provides a good estimate of the probability in many cases, it provides poor estimates when $n_c$ is very small. To see the difficulty, imagine that, in fact, the value of $P(Wind = strong | PlayTennis = no)$ is .08 and that we have a sample containing only 5 examples for which $PlayTennis = no$. Then the most probable value for $n_c$ is 0. This raises two difficulties. First, $\frac{n_c}{n}$ produces a biased underestimate of the probability. Second, when this probability estimate is zero, this probability term will dominate the Bayes classifier if the future query contains $Wind = strong$. The reason is that the quantity calculated in Equation (6.20) requires multiplying all the other probability terms by this zero value.

To avoid this difficulty we can adopt a Bayesian approach to estimating the probability, using the $m$-estimate defined as follows.

**$m$-estimate of probability:**

$$\frac{n_c + mp}{n + m} \tag{6.22}$$

Here, $n_c$ and $n$ are defined as before, $p$ is our prior estimate of the probability we wish to determine, and $m$ is a constant called the *equivalent sample size,* which determines how heavily to weight $p$ relative to the observed data. A typical method for choosing $p$ in the absence of other information is to assume uniform

priors; that is, if an attribute has $k$ possible values we set $p = \frac{1}{k}$. For example, in estimating $P(Wind = strong | PlayTennis = no)$ we note the attribute $Wind$ has two possible values, so uniform priors would correspond to choosing $p = .5$. Note that if $m$ is zero, the $m$-estimate is equivalent to the simple fraction $\frac{n_c}{n}$. If both $n$ and $m$ are nonzero, then the observed fraction $\frac{n_c}{n}$ and prior $p$ will be combined according to the weight $m$. The reason $m$ is called the equivalent sample size is that Equation (6.22) can be interpreted as augmenting the $n$ actual observations by an additional $m$ virtual samples distributed according to $p$.

## 6.10 AN EXAMPLE: LEARNING TO CLASSIFY TEXT

To illustrate the practical importance of Bayesian learning methods, consider learning problems in which the instances are text documents. For example, we might wish to learn the target concept "electronic news articles that I find interesting," or "pages on the World Wide Web that discuss machine learning topics." In both cases, if a computer could learn the target concept accurately, it could automatically filter the large volume of online text documents to present only the most relevant documents to the user.

We present here a general algorithm for learning to classify text, based on the naive Bayes classifier. Interestingly, probabilistic approaches such as the one described here are among the most effective algorithms currently known for learning to classify text documents. Examples of such systems are described by Lewis (1991), Lang (1995), and Joachims (1996).

The naive Bayes algorithm that we shall present applies in the following general setting. Consider an instance space $X$ consisting of all possible *text documents* (i.e., all possible strings of words and punctuation of all possible lengths). We are given training examples of some unknown target function $f(x)$, which can take on any value from some finite set $V$. The task is to learn from these training examples to predict the target value for subsequent text documents. For illustration, we will consider the target function classifying documents as interesting or uninteresting to a particular person, using the target values $like$ and $dislike$ to indicate these two classes.

The two main design issues involved in applying the naive Bayes classifier to such text classification problems are first to decide how to represent an arbitrary text document in terms of attribute values, and second to decide how to estimate the probabilities required by the naive Bayes classifier.

Our approach to representing arbitrary text documents is disturbingly simple: Given a text document, such as this paragraph, we define an attribute for each word position in the document and define the value of that attribute to be the English word found in that position. Thus, the current paragraph would be described by 111 attribute values, corresponding to the 111 word positions. The value of the first attribute is the word "our," the value of the second attribute is the word "approach," and so on. Notice that long text documents will require a larger number of attributes than short documents. As we shall see, this will not cause us any trouble.

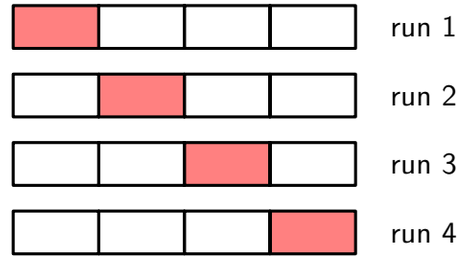



**Figure 1.18** The technique of $S$-fold cross-validation, illustrated here for the case of $S = 4$, involves taking the available data and partitioning it into $S$ groups (in the simplest case these are of equal size). Then $S - 1$ of the groups are used to train a set of models that are then evaluated on the remaining group. This procedure is then repeated for all $S$ possible choices for the held-out group, indicated here by the red blocks, and the performance scores from the $S$ runs are then averaged.

data to assess performance. When data is particularly scarce, it may be appropriate to consider the case $S = N$, where $N$ is the total number of data points, which gives the *leave-one-out* technique.

One major drawback of cross-validation is that the number of training runs that must be performed is increased by a factor of $S$, and this can prove problematic for models in which the training is itself computationally expensive. A further problem with techniques such as cross-validation that use separate data to assess performance is that we might have multiple complexity parameters for a single model (for instance, there might be several regularization parameters). Exploring combinations of settings for such parameters could, in the worst case, require a number of training runs that is exponential in the number of parameters. Clearly, we need a better approach. Ideally, this should rely only on the training data and should allow multiple hyperparameters and model types to be compared in a single training run. We therefore need to find a measure of performance which depends only on the training data and which does not suffer from bias due to over-fitting.

Historically various 'information criteria' have been proposed that attempt to correct for the bias of maximum likelihood by the addition of a penalty term to compensate for the over-fitting of more complex models. For example, the *Akaike information criterion*, or AIC (Akaike, 1974), chooses the model for which the quantity

$$\ln p(\mathcal{D}|\mathbf{w}_{\mathrm{ML}}) - M \tag{1.73}$$

is largest. Here $p(\mathcal{D}|\mathbf{w}_{\mathrm{ML}})$ is the best-fit log likelihood, and $M$ is the number of adjustable parameters in the model. A variant of this quantity, called the *Bayesian information criterion*, or *BIC*, will be discussed in Section 4.4.1. Such criteria do not take account of the uncertainty in the model parameters, however, and in practice they tend to favour overly simple models. We therefore turn in Section 3.4 to a fully Bayesian approach where we shall see how complexity penalties arise in a natural and principled way.

## 1.4. The Curse of Dimensionality

In the polynomial curve fitting example we had just one input variable $x$. For practical applications of pattern recognition, however, we will have to deal with spaces

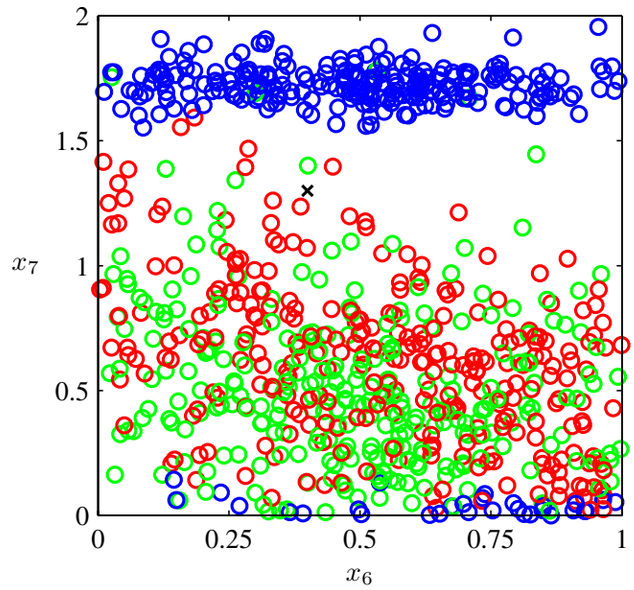



**Figure 1.19** Scatter plot of the oil flow data for input variables $x_6$ and $x_7$, in which red denotes the 'homogenous' class, green denotes the 'annular' class, and blue denotes the 'laminar' class. Our goal is to classify the new test point denoted by '$\times$'.

of high dimensionality comprising many input variables. As we now discuss, this poses some serious challenges and is an important factor influencing the design of pattern recognition techniques.

In order to illustrate the problem we consider a synthetically generated data set representing measurements taken from a pipeline containing a mixture of oil, water, and gas (Bishop and James, 1993). These three materials can be present in one of three different geometrical configurations known as 'homogenous', 'annular', and 'laminar', and the fractions of the three materials can also vary. Each data point comprises a 12-dimensional input vector consisting of measurements taken with gamma ray densitometers that measure the attenuation of gamma rays passing along narrow beams through the pipe. This data set is described in detail in Appendix A. Figure 1.19 shows 100 points from this data set on a plot showing two of the measurements $x_6$ and $x_7$ (the remaining ten input values are ignored for the purposes of this illustration). Each data point is labelled according to which of the three geometrical classes it belongs to, and our goal is to use this data as a training set in order to be able to classify a new observation $(x_6, x_7)$, such as the one denoted by the cross in Figure 1.19. We observe that the cross is surrounded by numerous red points, and so we might suppose that it belongs to the red class. However, there are also plenty of green points nearby, so we might think that it could instead belong to the green class. It seems unlikely that it belongs to the blue class. The intuition here is that the identity of the cross should be determined more strongly by nearby points from the training set and less strongly by more distant points. In fact, this intuition turns out to be reasonable and will be discussed more fully in later chapters.

How can we turn this intuition into a learning algorithm? One very simple approach would be to divide the input space into regular cells, as indicated in Figure 1.20. When we are given a test point and we wish to predict its class, we first decide which cell it belongs to, and we then find all of the training data points that

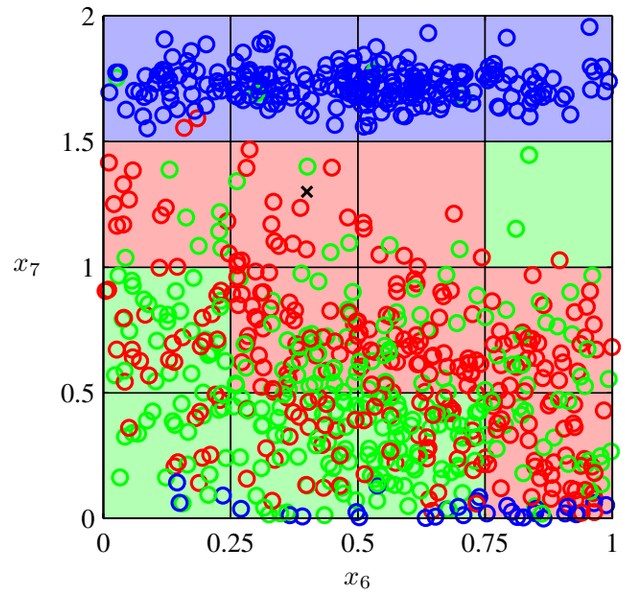



**Figure 1.20** Illustration of a simple approach to the solution of a classification problem in which the input space is divided into cells and any new test point is assigned to the class that has a majority number of representatives in the same cell as the test point. As we shall see shortly, this simplistic approach has some severe shortcomings.

fall in the same cell. The identity of the test point is predicted as being the same as the class having the largest number of training points in the same cell as the test point (with ties being broken at random).

There are numerous problems with this naive approach, but one of the most severe becomes apparent when we consider its extension to problems having larger numbers of input variables, corresponding to input spaces of higher dimensionality. The origin of the problem is illustrated in Figure 1.21, which shows that, if we divide a region of a space into regular cells, then the number of such cells grows exponentially with the dimensionality of the space. The problem with an exponentially large number of cells is that we would need an exponentially large quantity of training data in order to ensure that the cells are not empty. Clearly, we have no hope of applying such a technique in a space of more than a few variables, and so we need to find a more sophisticated approach.

We can gain further insight into the problems of high-dimensional spaces by returning to the example of polynomial curve fitting and considering how we would

*Section 1.1*

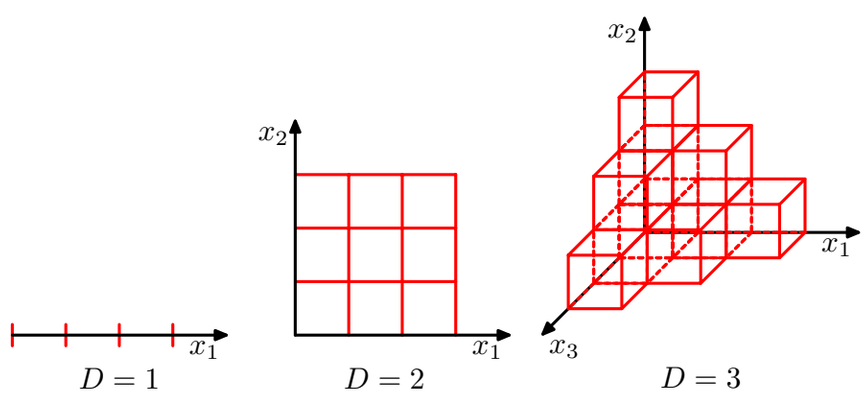



**Figure 1.21** Illustration of the curse of dimensionality, showing how the number of regions of a regular grid grows exponentially with the dimensionality $D$ of the space. For clarity, only a subset of the cubical regions are shown for $D = 3$.

CHAPTER 10

# Features

PREVIOUSLY I REFERRED to features as 'the workhorses of machine learning' – it is therefore high time to consider them in more detail. Features, also called attributes, are defined as mappings $f_i : \mathscr{X} \to \mathscr{F}_i$ from the instance space $\mathscr{X}$ to the feature domain $\mathscr{F}_i$. We can distinguish features by their domain: common feature domains include real and integer numbers, but also discrete sets such as colours, the Booleans, and so on. We can also distinguish features by the range of permissible operations. For example, we can calculate a group of people's average age but not their average blood type, so taking the average value is an operation that is permissible on some features but not on others. We will take a closer look at different kinds of feature in Section 10.1.

Although many data sets come with pre-defined features, they can be manipulated in many ways. For example, we can change the domain of a feature by rescaling or discretisation; we can select the best features from a larger set and only work with the selected ones; or we can combine two or more features into a new feature. In fact, a model itself is a way of constructing a new feature that solves the task at hand. Feature transformations will be investigated in Section 10.2, while feature construction and selection is the topic of Section 10.3

## 10.1 Kinds of feature

Consider two features, one describing a person's age and the other their house number. Both features map into the integers, but the way we use those features can be quite different. Calculating the average age of a group of people is meaningful, but an average house number is probably not very useful! In other words, what matters is not just the domain of a feature, but also the range of permissible operations. These, in turn, depend on whether the feature values are expressed on a meaningful *scale*. Despite appearances, house numbers are not really integers but *ordinals*: we can use them to determine that number 10's neighbours are number 8 and number 12, but we cannot assume that the distance between 8 and 10 is the same as the distance between 10 and 12. Because of the absence of a linear scale it is not meaningful to add or subtract house numbers, which precludes operations such as averaging.

### Calculations on features

Let's take a closer look at the range of possible calculations on features, often referred to as aggregates or statistics. Three main categories are *statistics of central tendency*, *statistics of dispersion* and *shape statistics*. Each of these can be interpreted either as a theoretical property of an unknown population or a concrete property of a given sample – here we will concentrate on sample statistics.

Starting with statistics of central tendency, the most important ones are

- the *mean* or average value;
- the *median*, which is the middle value if we order the instances from lowest to highest feature value; and
- the *mode*, which is the majority value or values.

Of these statistics, the mode is the one we can calculate whatever the domain of the feature: so, for example, we can say that the most frequent blood type in a group of people is O+. In order to calculate the median, we need to have an ordering on the feature values: so we can calculate both the mode and the median house number in a set of addresses.[1] In order to calculate the mean, we need a feature expressed on some scale: most often this will be a linear scale for which we calculate the familiar arithmetic mean, but Background 10.1 discusses means for some other scales. It is often suggested that the median tends to lie between the mode and the mean, but there are plenty of exceptions to this 'rule'. The famous statistician Karl Pearson suggested a

[1] If our sample contains an even number of instances, there are two middle values. If the feature has a scale it is customary to take the mean of those two values as the median; if the feature doesn't have a scale, or if it is important that we select a value actually occurring in the sample, we can either select both as the lower and upper median, or we can make a random choice.

Imagine a swimmer who swims the same distance $d$ on two different days, taking $a$ seconds one day and $b$ seconds the next. On average, it took her therefore $c = (a+b)/2$ seconds, with an average speed of $d/c = 2d/(a+b)$. Notice how this average speed is *not* calculated as the normal or *arithmetic mean* of the speeds, which would yield $(d/a + d/b)/2$: to calculate average speed over a fixed distance we use a different mean called the *harmonic mean*. Given two numbers $x$ and $y$ (in our swimming example these are the speeds on either day, $d/a$ and $d/b$), the harmonic mean $h$ is defined as

$$h(x,y) = \frac{2}{1/x + 1/y} = \frac{2xy}{x+y}$$

Since $1/h(x,y) = (1/x + 1/y)/2$, we observe that calculating the harmonic mean on a scale with unit $u$ corresponds to calculating the arithmetic mean on the *reciprocal scale* with unit $1/u$. In the example, speed with fixed distance is expressed on a scale reciprocal to the time scale, and since we use the arithmetic mean to average time, we use the harmonic mean to average speed. (If we average speed over a fixed *time* interval this is expressed on the same scale as distance and thus we would use the arithmetic mean.)

A good example of where the harmonic mean is used in machine learning arises when we average precision and recall of a classifier. Remember that precision is the proportion of positive predictions that is correct ($prec = TP/(TP+FP)$), and recall is the proportion of positives that is correctly predicted ($rec = TP/(TP+FN)$). Suppose we first calculate the number of mistakes averaged over the classes: this is the arithmetic mean $Fm = (FP + FN)/2$. We can then derive

$$\frac{TP}{TP+Fm} = \frac{TP}{TP+(FP+FN)/2} = \frac{2TP}{(TP+FP)+(TP+FN)} = \frac{2}{1/prec + 1/rec}$$

We recognise the last term as the harmonic mean of precision and recall. Since the enumerator of both precision and recall is fixed, taking the arithmetic mean of the denominators corresponds to taking the harmonic mean of the ratios. In information retrieval this harmonic mean of precision and recall is very often used and called the *F-measure*.

Yet other means exist for other scales. In music, going from one note to a note one octave higher corresponds to doubling the frequency. So frequencies $f$ and $4f$ are two octaves apart, and it makes sense to take the octave in between with frequency $2f$ as their mean. This is achieved by the *geometric mean*, which is defined as $g(x,y) = \sqrt{xy}$. Since $\log\sqrt{xy} = (\log xy)/2 = (\log x + \log y)/2$ it follows that the geometric mean corresponds to the arithmetic mean on a logarithmic scale. All these means have in common that the mean of two values is an intermediate value, and that they can easily be extended to more than two values.

**Background 10.1.** On scales and means.

more specific rule of thumb (with therefore even more exceptions): the median tends to fall one-third of the way from mean to mode.

The second kind of calculation on features are statistics of dispersion or 'spread'. Two well-known statistics of dispersion are the *variance* or average squared deviation from the (arithmetic) mean, and its square root, the *standard deviation*. Variance and standard deviation essentially measure the same thing, but the latter has the advantage that it is expressed on the same scale as the feature itself. For example, the variance of the body weight in kilograms of a group of people is measured in $kg^2$ (kilograms-squared), whereas the standard deviation is measured in kilograms. The absolute difference between the mean and the median is never larger than the standard deviation – this is a consequence of *Chebyshev's inequality*, which states that at most $1/k^2$ of the values are more than $k$ standard deviations away from the mean.

A simpler dispersion statistic is the difference between maximum and minimum value, which is called the *range*. A natural statistic of central tendency to be used with the range is the *midrange point*, which is the mean of the two extreme values. These definitions assume a linear scale but can be adapted to other scales using suitable transformations. For example, for a feature expressed on a logarithmic scale, such as frequency, we would take the ratio of the highest and lowest frequency as the range, and the harmonic mean of these two extremes as the midrange point.

Other statistics of dispersion include *percentiles*. The $p$-th percentile is the value such that $p$ per cent of the instances fall below it. If we have 100 instances, the 80th percentile is the value of the 81st instance in a list of increasing values.[2] If $p$ is a multiple of 25 the percentiles are also called *quartiles*, and if it is a multiple of 10 the percentiles are also called *deciles*. Note that the 50th percentile, the 5th decile and the second quartile are all the same as the median. Percentiles, deciles and quartiles are special cases of *quantiles*. Once we have quantiles we can measure dispersion as the distance between different quantiles. For instance, the *interquartile range* is the difference between the third and first quartile (i.e., the 75th and 25th percentile).

---

**Example 10.1 (Percentile plot).** Suppose you are learning a model over an instance space of countries, and one of the features you are considering is the gross domestic product (GDP) per capita. Figure 10.1 shows a so-called *percentile plot* of this feature. In order to obtain the $p$-th percentile, you intersect the line $y = p$ with the dotted curve and read off the corresponding percentile on the $x$-axis. Indicated in the figure are the 25th, 50th and 75th percentile. Also indicated is the

[2] Similar to the median there are issues with non-integer ranks, and they can be dealt with in different ways; however, significant differences do not arise unless the sample size is very small.

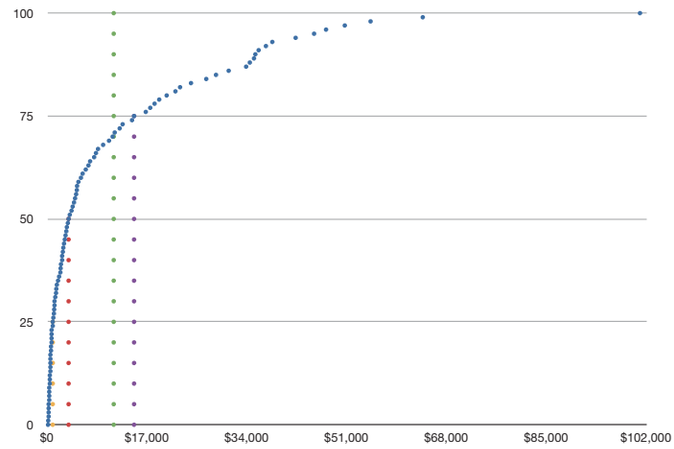



**Figure 10.1.** Percentile plot of GDP per capita for 231 countries (data obtained from `www.wolframalpha.com` by means of the query 'GDP per capita').The vertical dotted lines indicate, from left to right: the first quartile ($900); the median ($3600); the mean ($11 284); andthe third quartile ($14 750). The interquartile range is $13 850, while the standard deviation is $16 189.

mean (which has to be calculated from the raw data). As you can see, the mean is considerably higher than the median; this is mainly because of a few countries with very high GDP per capita. In other words, the mean is more sensitive to *outliers* than the median, which is why the median is often preferred to the mean for skewed distributions like this one.

---

You might think that the way I drew the percentile plot is the wrong way around: surely it would make more sense to have $p$ on the $x$-axis and the percentiles on the $y$-axis? One advantage of drawing the plot this way is that, by interpreting the $y$-axis as probabilities, the plot can be read as a *cumulative probability distribution*: a plot of $P(X \leq x)$ against $x$ for a random variable $X$. For example, the plot shows that $P(X \leq \mu)$ is approximately 0.70, where $\mu = \$11\,284$ is the mean GDP per capita. In other words, if you choose a random country the probability that its GDP per capita is less than the average is about 0.70.

Since GDP per capita is a real-valued feature, it doesn't necessarily make sense to talk about its mode, since if you measure the feature precisely enough every country will have a different value. We can get around this by means of a *histogram*, which counts the number of feature values in a particular interval or bin.

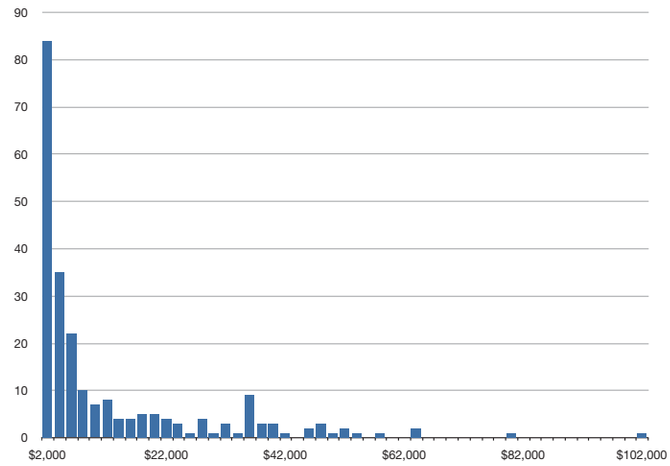



**Figure 10.2.** Histogram of the data from Figure 10.1, with bins of $2000 wide.

---

**Example 10.2 (Histogram).** A histogram of the data from Example 10.1 is shown in Figure 10.2. The left-most bin is the mode, with well over a third of the countries having a GDP per capita of not more than $2000. This demonstrates that the distribution is extremely *right-skewed* (i.e., has a long right tail), resulting in a mean that is considerably higher than the median.

---

The skew and 'peakedness' of a distribution can be measured by shape statistics such as skewness and kurtosis. The main idea is to calculate the third and fourth *central moment* of the sample. In general, the $k$-th central moment of a sample $\{x_i, \ldots, x_n\}$ is defined as $m_k = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^k$, where $\mu$ is the sample mean. Clearly, the first central moment is the average deviation from the mean – this is always zero, as the positive and negative deviations cancel each other out – and the second central moment is the average squared deviation from the mean, otherwise known as the variance. The third central moment $m_3$ can again be positive or negative. *Skewness* is then defined as $m_3/\sigma^3$, where $\sigma$ is the sample's standard deviation. A positive value of skewness means that the distribution is right-skewed, which means that the right tail is longer than the left tail. Negative skewness indicates the opposite, left-skewed case. *Kurtosis* is defined as $m_4/\sigma^4$. As it can be shown that a normal distribution has kurtosis 3, people often use *excess kurtosis* $m_4/\sigma^4 - 3$ as the statistic of interest. Briefly, positive excess kurtosis means that the distribution is more sharply peaked than the normal distribution.

| *Kind* | *Order* | *Scale* | *Tendency* | *Dispersion* | *Shape* |
|---|---|---|---|---|---|
| Categorical | × | × | mode | n/a | n/a |
| Ordinal | √ | × | median | quantiles | n/a |
| Quantitative | √ | √ | mean | range, interquartile range, variance, standard deviation | skewness, kurtosis |

**Table 10.1.** Kinds of feature, their properties and allowable statistics. Each kind inherits the statistics from the kinds above it in the table. For instance, the mode is a statistic of central tendency that can be computed for any kind of feature.

**Example 10.3 (Skewness and kurtosis).** In the GDP per capita example we can calculate skewness as 2.12 and excess kurtosis as 2.53. This confirms that the distribution is heavily right-skewed, and also more sharply peaked than the normal distribution.

## Categorical, ordinal and quantitative features

Given these various statistics we can distinguish three main kinds of feature: those with a meaningful numerical scale, those without a scale but with an ordering, and those without either. We will call features of the first type *quantitative*; they most often involve a mapping into the reals (another term in common use is 'continuous'). Even if a feature maps into a subset of the reals, such as age expressed in years, the various statistics such as mean or standard deviation still require the full scale of the reals.

Features with an ordering but without scale are called *ordinal features*. The domain of an ordinal feature is some totally ordered set, such as the set of characters or strings. Even if the domain of a feature is the set of integers, denoting the feature as ordinal means that we have to dispense with the scale, as we did with house numbers. Another common example are features that express a rank order: first, second, third, and so on. Ordinal features allow the mode and median as central tendency statistics, and quantiles as dispersion statistics.

Features without ordering or scale are called *categorical features* (or sometimes 'nominal' features). They do not allow any statistical summary except the mode. One subspecies of the categorical features is the *Boolean feature*, which maps into the truth values true and false. The situation is summarised in Table 10.1.

Models treat these different kinds of feature in distinct ways. First, consider tree models such as decision trees. A split on a categorical feature will have as many

children as there are feature values. Ordinal and quantitative features, on the other hand, give rise to a binary split, by selecting a value $v_0$ such that all instances with a feature value less than or equal to $v_0$ go to one child, and the remaining instances to the other child. It follows that tree models are insensitive to the scale of quantitative features. For example, whether a temperature feature is measured on the Celsius scale or on the Fahrenheit scale will not affect the learned tree. Neither will switching from a linear scale to a logarithmic scale have any effect: the split threshold will simply be $\log v_0$ instead of $v_0$. In general, tree models are insensitive to *monotonic* transformations on the scale of a feature, which are those transformations that do not affect the relative order of the feature values. In effect, *tree models ignore the scale of quantitative features, treating them as ordinal*. The same holds for rule models.

Now let's consider the naive Bayes classifier. We have seen that this model works by estimating a likelihood function $P(X|Y)$ for each feature $X$ given the class $Y$. For categorical and ordinal features with $k$ values this involves estimating $P(X = v_1|Y), \ldots, P(X = v_k|Y)$. In effect, ordinal features are treated as categorical ones, ignoring the order. Quantitative features cannot be handled at all, unless they are discretised into a finite number of bins and thus converted to categoricals. Alternatively, we could assume a parametric form for $P(X|Y)$, for instance a normal distribution. We will return to this later in this chapter when we discuss feature calibration.

While naive Bayes only really handles categorical features, many geometric models go in the other direction: they can only handle quantitative features. Linear models are a case in point: the very notion of linearity assumes a Euclidean instance space in which features act as Cartesian coordinates, and thus need to be quantitative. Distance-based models such as $k$-nearest neighbour and $K$-means require quantitative features if their distance metric is Euclidean distance, but we can adapt the distance metric to incorporate categorical features by setting the distance to 0 for equal values and 1 for unequal values (the ☞*Hamming distance* as defined in Section 8.1). In a similar vein, for ordinal features we can count the number of values between two feature values (if we encode the ordinal feature by means of integers, this would simply be their difference). This means that distance-based methods can accommodate all feature types by using an appropriate distance metric. Similar techniques can be used to extend support vector machines and other kernel-based methods to categorical and ordinal features.

### Structured features

It is usually tacitly assumed that an instance is a vector of feature values. In other words, the instance space is a Cartesian product of $d$ feature domains: $\mathscr{X} = \mathscr{F}_1 \times \ldots \times \mathscr{F}_d$. This means that there is no other information available about an instance apart from the information conveyed by its feature values. Identifying an instance with its

vector of feature values is what computer scientists call an *abstraction*, which is the result of filtering out unnecessary information. Representing an e-mail as a vector of word frequencies is an example of an abstraction.

However, sometimes it is necessary to avoid such abstractions, and to keep more information about an instance than can be captured by a finite vector of feature values. For example, we could represent an e-mail as a long string; or as a sequence of words and punctuation marks; or as a tree that captures the HTML mark-up; and so on. Features that operate on such structured instance spaces are called *structured features*.

---

**Example 10.4 (Structured features).** Suppose an e-mail is represented as a sequence of words. This allows us to define, apart from the usual word frequency features, a host of other features, including:

- ☞ whether the phrase 'machine learning' – or any other set of consecutive words – occurs in the e-mail;
- ☞ whether the e-mail contains at least eight consecutive words in a language other than English;
- ☞ whether the e-mail is palindromic, as in 'Degas, are we not drawn onward, we freer few, drawn onward to new eras aged?'

Furthermore, we could go beyond properties of single e-mails and express relations such as whether one e-mail is quoted in another e-mail, or whether two e-mails have one or more passages in common.

---

Structured features are not unlike *queries* in a database query language such as SQL or a declarative programming language such as Prolog. In fact, we have already seen examples of structured features in Section 6.4 when we looked at learning Prolog clauses such as the following:

```
fish(X):-bodyPart(X,Y).
fish(X):-bodyPart(X,pairOf(Z)).
```

The first clause has a single structured feature in the body which tests for the existence of some unspecified body part, while the second clause has another structured feature testing for the existence of a pair of unspecified body parts. The defining characteristic of structured features is that they involve *local variables* that refer to objects other than the instance itself. In a logical language such as Prolog it is natural to interpret local variables as existentially quantified, as we just did. However, it is equally possible to use other forms of *aggregation* over local variables: e.g., we can count the number of body parts (or pairs of body parts) an instance has.

| ↓ to, from → | *Quantitative* | *Ordinal* | *Categorical* | *Boolean* |
|---|---|---|---|---|
| *Quantitative* | **normalisation** | **calibration** | **calibration** | **calibration** |
| *Ordinal* | **discretisation** | **ordering** | **ordering** | **ordering** |
| *Categorical* | **discretisation** | **unordering** | **grouping** | |
| *Boolean* | **thresholding** | **thresholding** | **binarisation** | |

**Table 10.2.** An overview of possible feature transformations. **Normalisation and calibration** adapt the scale of quantitative features, or add a scale to features that don't have one. **Ordering** adds or adapts the order of feature values without reference to a scale. The other operations abstract away from unnecessary detail, either in a deductive way (**unordering**, **binarisation**) or by introducing new information (**thresholding**, **discretisation**).

Structured features can be constructed either prior to learning a model, or simultaneously with it. The first scenario is often called *propositionalisation* because the features can be seen as a translation from first-order logic to propositional logic without local variables. The main challenge with propositionalisation approaches is how to deal with combinatorial explosion of the number of potential features. Notice that features can be logically related: e.g., the second clause above covers a subset of the instances covered by the first one. It is possible to exploit this if structured feature construction is integrated with model building, as in inductive logic programming.

## 10.2 Feature transformations

*Feature transformations* aim at improving the utility of a feature by removing, changing or adding information. We could order feature types by the amount of detail they convey: quantitative features are more detailed than ordinal ones, followed by categorical features, and finally Boolean features. The best-known feature transformations are those that turn a feature of one type into another of the next type down this list. But there are also transformations that change the scale of quantitative features, or add a scale (or order) to ordinal, categorical and Boolean features. Table 10.2 introduces the terminology we will be using.

The simplest feature transformations are entirely deductive, in the sense that they achieve a well-defined result that doesn't require making any choices. *Binarisation* transforms a categorical feature into a set of Boolean features, one for each value of the categorical feature. This loses information since the values of a single categorical feature are mutually exclusive, but is sometimes needed if a model cannot handle more than two feature values. *Unordering* trivially turns an ordinal feature into a categorical one by discarding the ordering of the feature values. This is often required since most learning models cannot handle ordinal features directly. An interesting alternative that

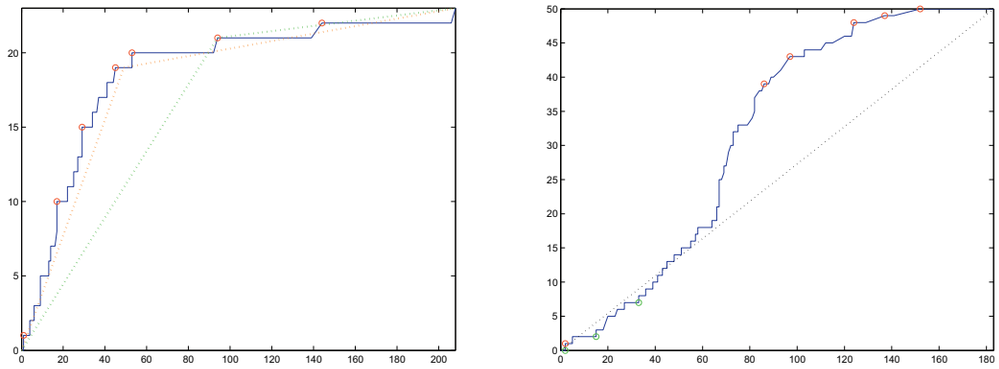



**Figure 10.3. (left)** Coverage curve obtained by ranking countries on decreasing GDP per capita, using 23 Euro countries as the positive class. The orange split sets the threshold equal to the mean, selecting 19 Euro countries and 49 non-Euro countries. The green split sets the threshold equal to the median, selecting 21 Euro countries and 94 non-Euro countries. The red points are on the convex hull of the coverage curve and indicate potentially optimal splits when the class label is taken into account. **(right)** Coverage curve of the same feature, using 50 countries in the Americas as the positive class. The red splits indicate potentially optimal thresholds with relatively many positives above the threshold, while the green splits indicate potentially optimal thresholds with relatively many positives below the threshold.

we will explore below is to add a scale to the feature by means of calibration.

In the remainder of this section we consider feature transformations that add information, the most important of which are discretisation and calibration.

## *Thresholding and discretisation*

*Thresholding* transforms a quantitative or an ordinal feature into a Boolean feature by finding a feature value to split on. I briefly alluded to this in Chapter 5 as a way to split on quantitative features in decision trees. Concretely, let $f : \mathscr{X} \to \mathbb{R}$ be a quantitative feature and let $t \in \mathbb{R}$ be a threshold, then $f_t : \mathscr{X} \to \{\mathsf{true}, \mathsf{false}\}$ is a Boolean feature defined by $f_t(x) = \mathsf{true}$ if $f(x) \geq t$ and $f_t(x) = \mathsf{false}$ if $f(x) < t$. We can choose such thresholds in an unsupervised or a supervised way.

---

**Example 10.5 (Unsupervised and supervised thresholding).** Consider the GDP per capita feature plotted in Figure 10.1 again. Without knowing how this feature is to be used in a model, the most sensible thresholds are the statistics of central tendency such as the mean and the median. This is referred to as *unsupervised thresholding*.

In a *supervised* learning setting we can do more. For example, suppose we want to use the GDP per capita as a feature in a decision tree to predict whether a country is one of the 23 countries that use the Euro as their official currency (or as one of their currencies). Using the feature as a ranker, we can construct a coverage curve (Figure 10.3 (left)). We see that for this feature the mean is not the most obvious threshold, as it splits right in the middle of a run of negatives. A better split is obtained at the start of that run of negatives, or at the end of the following run of positives, indicated by the red points at either end of the mean split. More generally, any point on the convex hull of the coverage curve represents a candidate threshold; which one to choose is informed by whether we put more value on picking out positives or negatives. As it happens in this example, the median threshold is on the convex hull, but this cannot be guaranteed in general as, by definition, unsupervised thresholding methods select the threshold independently from the target.

Figure 10.3 (right) shows the same feature with a different target: whether a country is in the Americas. We see that part of the curve is below the ascending diagonal, indicating that, in comparison with the whole data set, the initial segment of the ranking contains a smaller proportion of American countries. This means that potentially useful thresholds can also be found on the *lower convex hull*.

---

In summary, unsupervised thresholding typically involves calculating some statistic over the data, whereas supervised thresholding requires sorting the data on the feature value and traversing down this ordering to optimise a particular objective function such as information gain. Non-optimal split points could be filtered out by means of constructing the upper and lower convex hull, but in practice this is unlikely to be more efficient computationally than a straightforward sweep over the sorted instances.

If we generalise thresholding to multiple thresholds we arrive at one of the most commonly used non-deductive feature transformations. *Discretisation* transforms a quantitative feature into an ordinal feature. Each ordinal value is referred to as a *bin* and corresponds to an interval of the original quantitative feature. Again, we can distinguish between supervised and unsupervised approaches. *Unsupervised discretisation* methods typically require one to decide the number of bins beforehand. A simple method that often works reasonably well is to choose the bins so that each bin has approximately the same number of instances: this is referred to as *equal-frequency discretisation*. If we choose two bins then this method coincides with thresholding on the median. More generally, the bin boundaries are quantiles: for instance, with 10 bins the bin boundaries of equal-width discretisation are deciles. Another unsupervised

discretisation method is *equal-width discretisation*, which chooses the bin boundaries so that each interval has the same width. The interval width can be established by dividing the feature range by the number of bins if the feature has upper and lower limits; alternatively, we can take the bin boundaries at an integer number of standard deviations above and below the mean. An interesting alternative is to treat feature discretisation as a univariate clustering problem. For example, in order to generate $K$ bins we can uniformly sample $K$ initial bin centres and run $K$-means until convergence. We can alternatively use any of the other clustering methods discussed in Chapter 8: $K$-medoids, partitioning around medoids and hierarchical agglomerative clustering.

Switching now to *supervised discretisation* methods, we can distinguish between *top–down* or *divisive* discretisation methods on the one hand, and *bottom–up* or *agglomerative* discretisation methods on the other. Divisive methods work by progressively splitting bins, whereas agglomerative methods proceed by initially assigning each instance to its own bin and successively merging bins. In either case an important role is played by the *stopping criterion*, which decides whether a further split or merge is worthwhile. We give an example of each strategy. A natural generalisation of thresholding leads to a top–down *recursive partitioning* algorithm (Algorithm 10.1). This discretisation algorithm finds the best threshold according to some scoring function $Q$, and proceeds to recursively split the left and right bins. One scoring function that is often used is information gain.

---

**Example 10.6 (Recursive partitioning using information gain).** Consider the following feature values, which are ordered on increasing value for convenience.

| *Instance* | *Value* | *Class* |
|---|---|---|
| $e_1$ | −5.0 | ⊖ |
| $e_2$ | −3.1 | ⊕ |
| $e_3$ | −2.7 | ⊖ |
| $e_4$ | 0.0 | ⊖ |
| $e_5$ | 7.0 | ⊖ |
| $e_6$ | 7.1 | ⊕ |
| $e_7$ | 8.5 | ⊕ |
| $e_8$ | 9.0 | ⊖ |
| $e_9$ | 9.0 | ⊕ |
| $e_{10}$ | 13.7 | ⊖ |
| $e_{11}$ | 15.1 | ⊖ |
| $e_{12}$ | 20.1 | ⊖ |

This feature gives rise to the following ranking: ⊖⊕⊖⊖⊖⊕⊕[⊖⊕]⊖⊖⊖, where the square brackets indicate a tie between instances $e_8$ and $e_9$. The corresponding

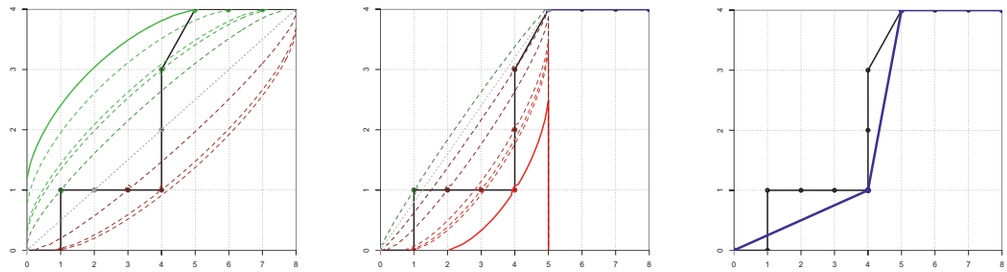


**Figure 10.4. (left)** A coverage curve visualising the ranking of four positive and eight negative examples by a feature to be discretised. The curved lines are information gain isometrics through possible split points; the solid isometric indicates the best split $[4+,5-][0+,3-]$ according to information gain. **(middle)** Recursive partitioning proceeds to split the segment $[4+,5-]$ into $[1+,4-][3+,1-]$. **(right)** If we stop here, the blue curve visualises the discretised (but still ordinal) feature.

coverage curve is depicted in Figure 10.4. Tracing information gain isometrics through each possible split, we see that the best split is ⊖⊕⊖⊖⊖⊕⊕[⊖⊕]|⊖⊖⊖. Repeating the process once more gives the discretisation ⊖⊕⊖⊖⊖|⊕⊕[⊖⊕]|⊖⊖⊖.

Clearly, we can stop the recursive partitioning algorithm when the empirical probabilities are the same across the ranking; this has pure bins and bins with a constant feature value as special cases. With this stopping criterion, the algorithm will successfully identify all straight line segments in the ranking. In fact, it is not hard to see that this holds true even if we change the scoring function – the split points may be found in a different order, but the end result will be the same. In practice more aggressive stopping criteria are used, which does mean that the end result depends on the

**Algorithm 10.1:** RecPart($S, f, Q$) – supervised discretisation by means of recursive partitioning.

**Input** : set of labelled instances $S$ ranked on feature values $f(x)$; scoring function $Q$.

**Output** : sequence of thresholds $t_1, \ldots, t_{k-1}$.

```
if stopping criterion applies then return ∅;
Split S into S_l and S_r using threshold t that optimises Q ;
T_l = RecPart(S_l, f, Q);
T_r = RecPart(S_r, f, Q);
return T_l ∪ {t} ∪ T_r;
```

**Algorithm 10.2:** AggloMerge($S, f, Q$) – supervised discretisation by means of agglomerative merging.

**Input** : set of labelled instances $S$ ranked on feature values $f(x)$; scoring function $Q$.
**Output** : sequence of thresholds.

```
initialise bins to data points with the same scores;
merge consecutive pure bins ;                    // optional optimisation
repeat
    evaluate Q on consecutive bin pairs;
    merge the pairs with best Q (unless they invoke the stopping criterion);
until no further merges are possible;
return thresholds between bins;
```

scoring function. For example, in Figure 10.4 we see that the split ⊖|[⊕⊖]⊕⊕⊖⊖⊖⊕⊕ has the second-highest information gain but ends up not being chosen at all, while with a different scoring function it might have been chosen in the first round. One of the most popular stopping criteria applies a minimum description length argument to decide whether a given bin should be split further.

It should be noted that the data set in Example 10.6 is probably so small that the stopping criterion will kick in straight away and recursive partitioning will be unable to go beyond a single bin. More generally, this kind of discretisation tends to be fairly conservative. For example, on the Euro data in Figure 10.3 (left) recursive partitioning produces two bins, selecting 20 Euro countries and 53 non-Euro countries (the red point in between the mean and median splits). On the American countries data in Figure 10.3 (right) we again obtain two bins, corresponding to the third red point from the right.

An algorithm for bottom–up *agglomerative merging* is given in Algorithm 10.2. Again the algorithm can take various choices for the scoring function and the stopping criterion: a popular choice is to use the $\chi^2$ *statistic* for both.

---

**Example 10.7 (Agglomerative merging using $\chi^2$).** We continue Example 10.6. Algorithm 10.2 initialises the bins to ⊖|⊕|⊖⊖⊖|⊕⊕|[⊖⊕]|⊖⊖⊖. We illustrate the calculation of the $\chi^2$ statistic for the last two bins. We construct the following contingency table:

| | *Left bin* | *Right bin* | |
|---|---|---|---|
| ⊕ | **1** | **0** | 1 |
| ⊖ | **1** | **3** | 4 |
| | 2 | 3 | 5 |

At the basis of the $\chi^2$ statistic lies a comparison of these observed frequencies with expected frequencies obtained from the row and column marginals. For example, the marginals say that the top row contains 20% of the total mass and the left column 40%; so if rows and columns were statistically independent we would expect 8% of the mass – or 0.4 of the five instances – in the top-left cell. Following a clockwise direction, the expected frequencies for the other cells are 0.6, 2.4 and 1.6. If the observed frequencies are close to the expected ones, this suggests that these two bins are candidates for merging since the split appears to have no bearing on the class distribution.

The $\chi^2$ statistic sums the squared differences between the observed and expected frequencies, each term normalised by the expected frequency:

$$\chi^2 = \frac{(1-0.4)^2}{0.4} + \frac{(0-0.6)^2}{0.6} + \frac{(3-2.4)^2}{2.4} + \frac{(1-1.6)^2}{1.6} = 1.88$$

Going left-to-right through the other pairs of consecutive bins, the $\chi^2$ values are 2, 4, 5 and 1.33 (there's an easy way to calculate the $\chi^2$ value for two pure bins, which I'll leave you to discover). This tells us that the fourth and fifth bin are first to be merged, leading to ⊖|⊕|⊖⊖⊖|⊕⊕[⊖⊕]|⊖⊖⊖. We then recompute the $\chi^2$ values ( in fact, only those involving the newly merged bin need to be re-computed), yielding 2, 4, 3.94 and 3.94. We now merge the first two bins, giving the partition ⊖⊕|⊖⊖⊖|⊕⊕[⊖⊕]|⊖⊖⊖. This changes the first $\chi^2$ value to 1.88, so we again merge the first two bins, arriving at ⊖⊕⊖⊖⊖|⊕⊕[⊖⊕]|⊖⊖⊖ (the same three bins as in Example 10.6).

---

In agglomerative discretisation the stopping criterion usually takes the form of a simple threshold on the scoring function. In the case of the $\chi^2$ statistic, the threshold can be derived from the $p$-value associated with the $\chi^2$ distribution, which is the probability of observing a $\chi^2$ value above the threshold if the two variables are actually independent. For two classes (i.e., one degree of freedom) and a $p$-value of 0.10 the $\chi^2$ threshold is 2.71, which in our example means that we stop at the above three bins. For a lower $p$-value of 0.05 the $\chi^2$ threshold is 3.84, which means that we eventually merge all the bins.

Notice that both top–down and bottom–up supervised discretisation bear some resemblance to algorithms we have seen previously: recursive partitioning shares the divide-and-conquer nature of the ☞*decision tree* training algorithm (Algorithm 5.1 on p.132), and agglomerative discretisation by merging consecutive bins is related to ☞*hierarchical agglomerative clustering* (Algorithm 8.4 on p.255). It is also worth mentioning that, although our examples were predominantly drawn from binary classification, most methods can handle more than two classes without complication.

## Normalisation and calibration

Thresholding and discretisation are feature transformations that remove the scale of a quantitative feature. We now turn our attention to adapting the scale of a quantitative feature, or adding a scale to an ordinal or categorical feature. If this is done in an unsupervised fashion it is usually called normalisation, whereas calibration refers to supervised approaches taking in the (usually binary) class labels. *Feature normalisation* is often required to neutralise the effect of different quantitative features being measured on different scales. If the features are approximately normally distributed, we can convert them into ☞*z-scores* (Background 9.1 on p.267) by centring on the mean and dividing by the standard deviation. In certain cases it is mathematically more convenient to divide by the variance instead, as we have seen in Section 7.1. If we don't want to assume normality we can centre on the median and divide by the interquartile range.

Sometimes feature normalisation is understood in the stricter sense of expressing the feature on a $[0,1]$ scale. This can be achieved in various ways. If we know the feature's highest and lowest values $h$ and $l$, then we can simply apply the linear scaling $f \mapsto (f-l)/(h-l)$. We sometimes have to guess the value of $h$ or $l$, and truncate any value outside $[l,h]$. For example, if the feature measures age in years, we may take $l = 0$ and $h = 100$, and truncate any $f > h$ to 1. If we can assume a particular distribution for the feature, then we can work out a transformation such that almost all feature values fall in a certain range. For instance, we know that more than 99% of the probability mass of a normal distribution falls within $\pm 3\sigma$ of the mean, where $\sigma$ is the standard deviation, so the linear scaling $f \mapsto (f-\mu)/6\sigma + 1/2$ virtually removes the need for truncation.

*Feature calibration* is understood as a supervised feature transformation adding a meaningful scale carrying class information to arbitrary features. This has a number of important advantages. For instance, it allows models that require scale, such as linear classifiers, to handle categorical and ordinal features. It also allows the learning algorithm to choose whether to treat a feature as categorical, ordinal or quantitative. We will assume a binary classification context, and so a natural choice for the calibrated feature's scale is the posterior probability of the positive class, conditioned on

the feature's value. This has the additional advantage that models that are based on such probabilities, such as naive Bayes, do not require any additional training once the features are calibrated, as we shall see. The problem of feature calibration can thus be stated as follows: given a feature $F : \mathscr{X} \to \mathscr{F}$, construct a calibrated feature $F^{\mathrm{c}} : \mathscr{X} \to [0,1]$ such that $F^{\mathrm{c}}(x)$ estimates the probability $F^{\mathrm{c}}(x) = P(\oplus|v)$, where $v = F(x)$ is the value of the original feature for $x$.

For categorical features this is as straightforward as collecting relative frequencies from a training set.

---

**Example 10.8 (Calibration of categorical features).** Suppose we want to predict whether or not someone has diabetes from categorical features including whether the person is obese or not, whether he or she smokes, and so on. We collect some statistics which tell us that 1 in every 18 obese persons has diabetes while among non-obese people this is 1 in 55 (data obtained from `www.wolframalpha.com` with the query 'diabetes'). If $F(x) = 1$ for person $x$ who is obese and $F(y) = 0$ for person $y$ who isn't, then the calibrated feature values are $F^{\mathrm{c}}(x) = 1/18 = 0.055$ and $F^{\mathrm{c}}(y) = 1/55 = 0.018$.

---

In fact, it would be better to compensate for the non-uniform class distribution, in order to avoid over-emphasising the class prior, which is better taken into account in the decision rule. This can be achieved as follows. If $m$ of $n$ obese people have diabetes, then this corresponds to a posterior odds of $m/(n-m)$ or a likelihood ratio of $m/c(n-m)$, where $c$ is the prior odds of having diabetes (since posterior odds is likelihood ratio times prior odds). Working with the likelihood ratio is equivalent to assuming a uniform class distribution. Converting the likelihood ratio into a probability gives

$$F^{\mathrm{c}}(x) = \frac{\frac{m}{c(n-m)}}{\frac{m}{c(n-m)}+1} = \frac{m}{m+c(n-m)}$$

In our example, if the prior odds of having diabetes is $c = 1/48$, then $F^{\mathrm{c}}(x) = 1/(1+17/48) = 48/(48+17) = 0.74$. The extent to which this probability is more than 1/2 quantifies the extent to which obese people are more likely than average to have diabetes. For non-obese people the probability is $1/(1+54/48) = 48/(48+54) = 0.47$, so they are slightly less likely than average to have diabetes. Keep in mind also that it is usually a good idea to smooth these probability estimates by means of the Laplace correction, which adds 1 to $m$ and 2 to $n$. This leads to the final expression for calibrating a categorical feature:

$$F^{\mathrm{c}}(x) = \frac{m+1}{m+1+c(n-m+1)}$$

Ordinal and quantitative features can be discretised and then calibrated as categorical features. In the remainder of this section we look at calibration methods that maintain the ordering of the feature. For example, suppose we want to use body weight as an indicator for diabetes. A calibrated weight feature attaches a probability to every weight, such that these probabilities are non-decreasing with weight. This is related to our discussion of ☞*calibrating classifier scores* in Section 7.4, as those calibrated probabilities should likewise takes the ranking of the classifier's predictions into account. In fact, the two approaches to classifier calibration – by employing the logistic function and by constructing the ROC convex hull – are directly applicable to feature calibration, since a quantitative feature can simply be treated as a univariate scoring classifier.

We briefly reiterate the main points of *logistic calibration*, but with a slight change in notation. Let $F : \mathscr{X} \to \mathbb{R}$ be a quantitative feature with class means $\mu^{\oplus}$ and $\mu^{\ominus}$ and variance $\sigma^2$. Assuming the feature is normally distributed within each class with the same variance, we can express the likelihood ratio of a feature value $v$ as

$$\begin{aligned} LR(v) = \frac{P(v|\oplus)}{P(v|\ominus)} &= \exp\left(\frac{-(v-\mu^{\oplus})^2 + (v-\mu^{\ominus})^2}{2\sigma^2}\right) \\ &= \exp\left(\frac{\mu^{\oplus}-\mu^{\ominus}}{\sigma}\,\frac{v-(\mu^{\oplus}+\mu^{\ominus})/2}{\sigma}\right) = \exp\left(d'z\right) \end{aligned}$$

where $d' = (\mu^{\oplus} - \mu^{\ominus})/\sigma$ is the difference between the means in proportion to the standard deviation, which is known as *d-prime* in signal detection theory; and $z = (v-\mu)/\sigma$ is the $z$-score associated with $v$ (notice we take the mean as $\mu = (\mu^{\oplus} + \mu^{\ominus})/2$ to simulate an equal class distribution). Again we work directly with the likelihood ratio to neutralise the effect of a non-uniform class distribution, and we obtain the calibrated feature value as

$$F^{\mathrm{c}}(x) = \frac{LR(F(x))}{1+LR(F(x))} = \frac{\exp\left(d'z(x)\right)}{1+\exp\left(d'z(x)\right)}$$

You may recognise the logistic function we discussed in Chapter 7 (see Figure 7.11 on p.222).

In essence, logistic feature calibration performs the following steps.

1. Estimate the class means $\mu^{\oplus}$ and $\mu^{\ominus}$ and the standard deviation $\sigma$.
2. Transform $F(x)$ into $z$-scores $z(x)$, making sure to use $\mu = (\mu^{\oplus} + \mu^{\ominus}))/2$ as the feature mean.
3. Rescale the $z$-scores to $F^{\mathrm{d}}(x) = d'z(x)$ with $d' = (\mu^{\oplus} - \mu^{\ominus})/\sigma$.
4. Apply a sigmoidal transformation to $F^{\mathrm{d}}(x)$ to give calibrated probabilities $F^{\mathrm{c}}(x) = \frac{\exp(F^{\mathrm{d}}(x))}{1+\exp(F^{\mathrm{d}}(x))}$.

Sometimes it is preferred to work directly with $F^{\mathrm{d}}(x)$, as it is expressed on a scale linearly related to the original feature's scale, and the Gaussian assumption implies that

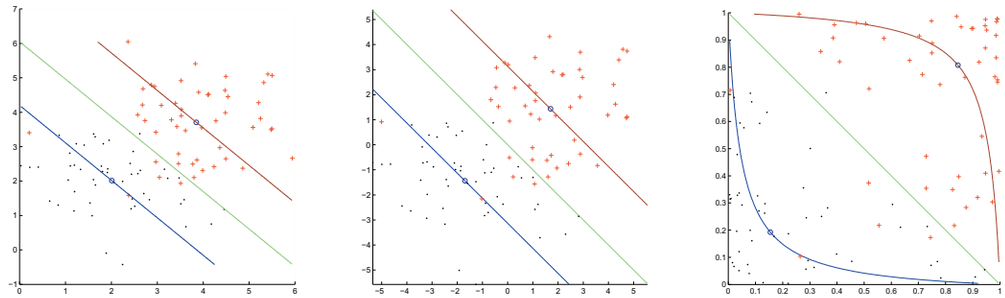


**Figure 10.5. (left)** Two-class Gaussian data. The middle line is the decision boundary learned by the basic linear classifier; the other two are parallel lines through the class means. **(middle)** Logistic calibration to log-odds space is a linear transformation; assuming unit standard deviations, the basic linear classifier is now the fixed line $F_1^{\mathrm{d}}(x) + F_2^{\mathrm{d}}(x) = 0$. **(right)** Logistic calibration to probability space is a non-linear transformation that pushes data away from the decision boundary.

we expect that scale to be additive. For example, distance-based models expect additive features in order to calculate Euclidean distance. In contrast, the scale of $F^{\mathrm{c}}$ is multiplicative. Notice that the two are interdefinable as $F^{\mathrm{d}}(x) = \ln \frac{F^{\mathrm{c}}(x)}{1-F^{\mathrm{c}}(x)} = \ln F^{\mathrm{c}}(x) - \ln(1 - F^{\mathrm{c}}(x))$. I will call the feature space spanned by $F^{\mathrm{d}}$ *log-odds space*, since $\exp\left(F^{\mathrm{d}}(x)\right) = LR(x)$ and the likelihood ratio is equal to the odds if we're assuming a uniform class prior. Calibrated features $F^{\mathrm{c}}$ live in *probability space*.

---

**Example 10.9 (Logistic calibration of two features).** Logistic feature calibration is illustrated in Figure 10.5. I generated two sets of 50 points by sampling bivariate Gaussians with identity covariance matrix, centred at (2,2) and (4,4). I then constructed the basic linear classifier as well as two parallel decision boundaries through the class means. Tracing these three lines in calibrated space will help us understand feature calibration.

In the middle figure we see the transformed data in log-odds space, which is clearly a linear rescaling of the axes. The basic linear classifier is now the line $F_1^{\mathrm{d}}(x) + F_2^{\mathrm{d}}(x) = 0$ through the origin. In other words, for this simple classifier feature calibration has removed the need for further training: instead of fitting a decision boundary to the data, we have fitted the data to a fixed decision boundary! (I should add that I cheated very slightly here, as I fixed $\sigma = 1$ in the calibration process – had I estimated each feature's standard deviation from the data, the decision boundary would most likely have had a slightly different slope.)

On the right we see the transformed data in probability space, which clearly has a non-linear relationship with the other two feature spaces. The basic linear

classifier is still linear in this space, but actually this is no longer true for more than two features. To see this, note that $F_1^c(x) + F_2^c(x) = 1$ can be rewritten as

$$\frac{\exp\left(F_1^d(x)\right)}{1+\exp\left(F_1^d(x)\right)} + \frac{\exp\left(F_2^d(x)\right)}{1+\exp\left(F_2^d(x)\right)} = 1$$

which can be simplified to $\exp\left(F_1^d(x)\right)\exp\left(F_2^d(x)\right) = 1$ and hence to $F_1^d(x) + F_2^d(x) = 0$. However, if we add a third feature not all cross-terms cancel and we obtain a non-linear boundary .

---

The log-odds representation does hold an interest in another respect. An arbitrary linear decision boundary in log-odds space is represented by $\sum_i w_i F_i^d(x) = t$. Taking natural logarithms this can be rewritten as

$$\exp\left(\sum_i w_i F_i^d(x)\right) = \prod_i \exp\left(w_i F_i^d(x)\right) = \prod_i \left(\exp\left(F_i^d(x)\right)\right)^{w_i} = \prod_i LR_i(x)^{w_i} = \exp(t) = t'$$

This exposes a connection with the naive Bayes models discussed in Section 9.2, whose decision boundaries are also defined as products of likelihood ratios for individual features. The basic naive Bayes model has $w_i = 1$ for all $i$ and $t' = 1$, which means that *fitting data to a fixed linear decision boundary in log-odds space by means of feature calibration can be understood as training a naive Bayes model*. Changing the slope of the decision boundary corresponds to introducing non-unit feature weights, which is similar to the way feature weights arose in the multinomial naive Bayes model.

It is instructive to investigate the distribution of the calibrated feature a bit more (I will omit the technical details). Assuming the uncalibrated distributions were two Gaussian bumps, what do the calibrated distributions look like? We have already seen that calibrated data points are pulled away from the decision boundary, so we would expect the peaks of the calibrated distributions to be closer to their extreme values. How much closer depends solely on $d'$; Figure 10.6 depicts the calibrated distributions for various values of $d'$.

We move on to *isotonic calibration*, a method that requires order but ignores scale and can be applied to both ordinal and quantitative features. We essentially use the feature as a univariate ranker, and construct its ROC curve and convex hull to obtain a piecewise-constant calibration map. Suppose we have an ROC curve, and suppose the $i$-th segment of the curve involves $n_i$ training examples, out of which $m_i$ are positives. The corresponding ROC segment has slope $l_i = m_i/(c(n_i - m_i))$, where $c$ is the prior odds. Suppose first the ROC curve is convex: i.e., $i < j$ implies $l_i \geq l_j$. In that case, we can use the same formula as for categorical features to obtain a calibrated feature

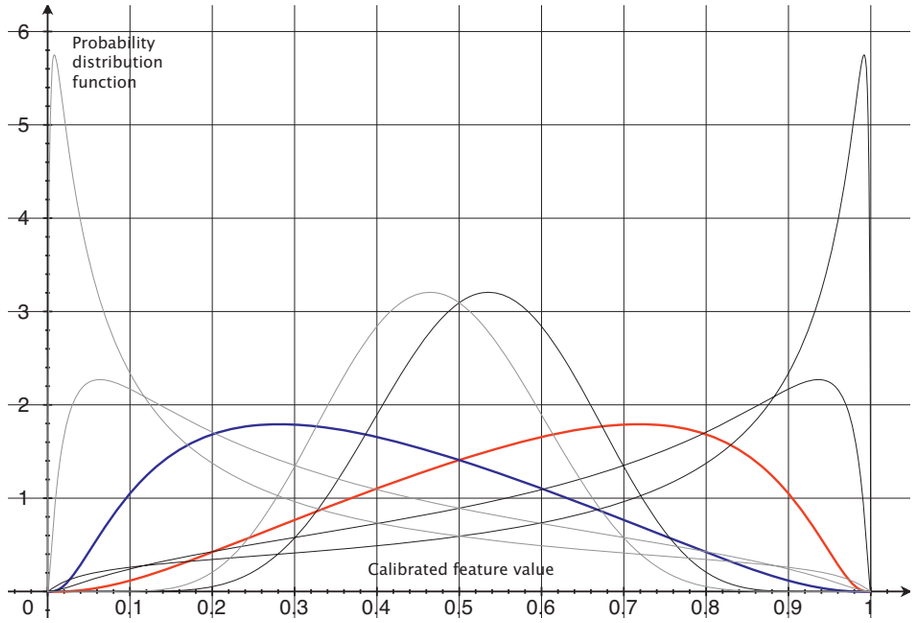



**Figure 10.6.** Per-class distributions of a logistically calibrated feature for different values of $d'$, the distance between the uncalibrated class means in proportion to the feature's standard deviation. The red and blue curves depict the distributions for the positive and negative class for a feature whose means are one standard deviation apart ($d' = 1$). The other curves are for $d' \in \{0.5, 1.4, 1.8\}$.

value:

$$v_i^{\mathrm{c}} = \frac{m_i + 1}{m_i + 1 + c(n_i - m_i + 1)} \tag{10.1}$$

As before, this achieves both probability smoothing through Laplace correction and compensation for non-uniform class distributions. If the ROC curve is not convex, there exist $i < j$ such that $l_i < l_j$. Assuming we want to maintain the original feature ordering, we first construct the convex hull of the ROC curve. The effect of this is that we join adjacent segments in the ROC curve that are part of a concavity, until no concavities remain. We recalculate the segments and assign calibrated feature values as in Equation 10.1.

---

**Example 10.10 (Isotonic feature calibration).** The following table gives sample values of a weight feature in relation to a diabetes classification problem. Figure 10.7 shows the ROC curve and convex hull of the feature and the calibration map obtained by isotonic calibration.

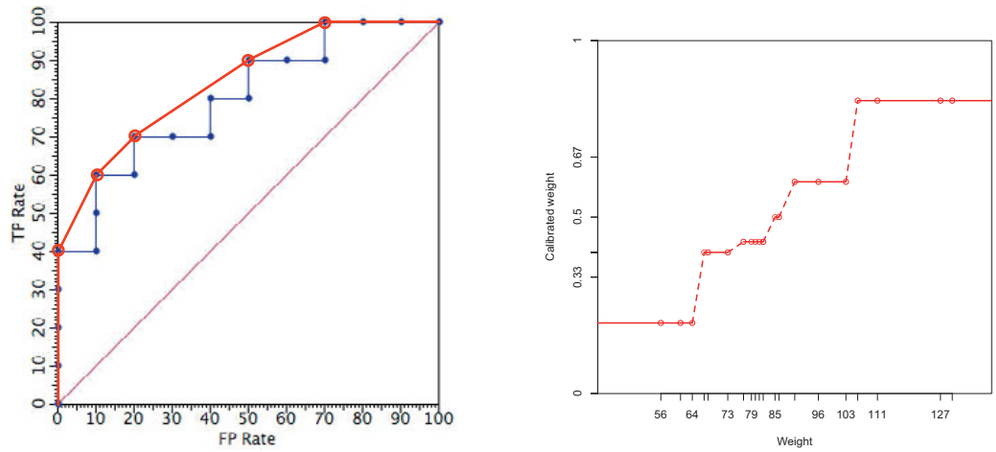



**Figure 10.7. (left)** ROC curve and convex hull of an uncalibrated feature. Calibrated feature values are obtained from the proportion of positives in each segment of the ROC convex hull. **(right)** The corresponding piecewise-constant calibration map, which maps the uncalibrated feature values on the $x$-axis to the calibrated feature values on the $y$-axis.

| *Weight* | *Diabetes?* | *Calibrated weight* | *Weight* | *Diabetes?* | *Calibrated weight* |
|---|---|---|---|---|---|
| 130 | ⊕ | 0.83 | 81 | ⊖ | 0.43 |
| 127 | ⊕ | 0.83 | 80 | ⊕ | 0.43 |
| 111 | ⊕ | 0.83 | 79 | ⊖ | 0.43 |
| 106 | ⊕ | 0.83 | 77 | ⊕ | 0.43 |
| 103 | ⊖ | 0.60 | 73 | ⊖ | 0.40 |
| 96 | ⊕ | 0.60 | 68 | ⊖ | 0.40 |
| 90 | ⊕ | 0.60 | 67 | ⊕ | 0.40 |
| 86 | ⊖ | 0.50 | 64 | ⊖ | 0.20 |
| 85 | ⊕ | 0.50 | 61 | ⊖ | 0.20 |
| 82 | ⊖ | 0.43 | 56 | ⊖ | 0.20 |

For example, a weight of 80 kilograms is calibrated to 0.43, meaning that three out of seven people in that weight interval have diabetes (after Laplace correction).

Example 10.11 gives a bivariate illustration. As is clearly visible, for quantitative features the process amounts to supervised discretisation of the feature values, which means that many points are mapped to the same point in calibrated space. This is different from logistic calibration, which is invertible.

**Example 10.11 (Isotonic calibration of two features).** Figure 10.8 shows the result of isotonic calibration on the same data as in Example 10.9, both in log-odds

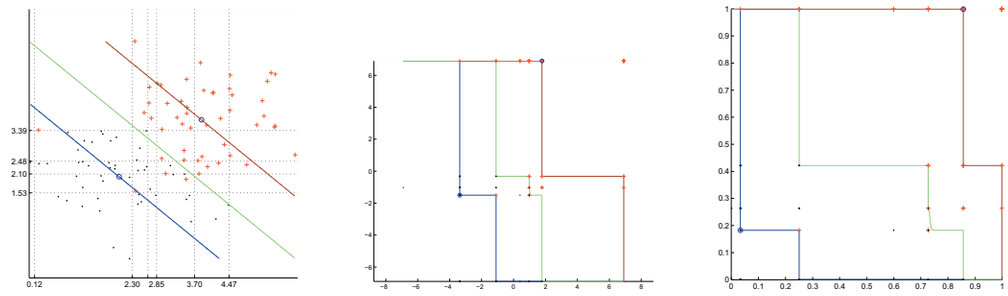



**Figure 10.8.** **(left)** The data from Figure 10.5, with grid lines indicating the discretisation obtained by isotonic feature calibration. **(middle)** Isotonically calibrated data in log-odds space. **(right)** Isotonically calibrated data in probability space.

space and in probability space. Because of the discrete nature of isotonic calibration, even the transformation to log-odds space is no longer linear: the basic linear classifier becomes a series of axis-parallel line segments. This is also true in the opposite direction: if we imagine a linear decision boundary in log-odds space or in probability space, this maps to a decision boundary following the dotted lines in the original feature space. Effectively, isotonic feature calibration has changed the linear grading model into a grouping model.

---

In summary, isotonic feature calibration performs the following steps.

1. Sort the training instances on feature value and construct the ROC curve. The sort order is chosen such that the ROC curve has AUC $\geq 1/2$.

2. Construct the convex hull of this curve, and count the number of positives $m_i$ and the total number of instances $n_i$ in each segment of the convex hull.

3. Discretise the feature according to the convex hull segments, and associate a calibrated feature value $v_i^{\mathrm{c}} = \frac{m_i+1}{m_i+1+c(n_i-m_i+1)}$ with each segment.

4. If an additive scale is required, use $v_i^{\mathrm{d}} = \ln \frac{v_i^{\mathrm{c}}}{1-v_i^{\mathrm{c}}} = \ln v_i^{\mathrm{c}} - \ln(1-v_i^{\mathrm{c}})$.

## Incomplete features

At the end of this section on feature transformations we briefly consider what to do if we don't know a feature's value for some of the instances. We encountered this situation in Example 1.2 on p.26, where we discussed how to classify an e-mail if we didn't know whether it contained one of the vocabulary words or not. Probabilistic models

handle this rather gracefully by taking a weighted average over all possible values of the feature:

$$P(Y|X) = \sum_z P(Y, Z = z|X) = \sum_z P(Y|X, Z = z)P(Z = z)$$

Here, $Y$ is the target variable as usual, $X$ stands for the features that are observed for the instance to be classified, while $Z$ are the features that are unobserved at classification time. The distribution $P(Z)$ can be obtained from the trained model, at least for a generative model – if our model is discriminative we need to estimate $P(Z)$ separately.

Missing feature values at training time are trickier to handle. First of all, the very fact that a feature value is missing may be correlated with the target variable. For example, the range of medical tests carried out on a patient is likely to depend on their medical history. For such features it may be best to have a designated 'missing' value so that, for instance, a tree model can split on it. However, this would not work for, say, a linear model. In such cases we can complete the feature by 'filling in' the missing values, a process known as *imputation*. For instance, in a classification problem we can calculate the per-class means, medians or modes over the observed values of the feature and use this to impute the missing values. A somewhat more sophisticated method takes feature correlation into account by building a predictive model for each incomplete feature and uses that model to 'predict' the missing value. It is also possible to invoke the ☞*Expectation-Maximisation* algorithm (Section 9.4), which goes roughly as follows: assuming a multivariate model over all features, use the observed values for maximum-likelihood estimation of the model parameters, then derive expectations for the unobserved feature values and iterate.

## 10.3 Feature construction and selection

The previous section on feature transformation makes it clear that there is a lot of scope in machine learning to play around with the original features given in the data. We can take this one step further by constructing new features from several original features. A simple example of this can be used to improve the ☞*naive Bayes* classifier discussed in Section 9.2. Remember that in text classification applications we have a feature for every word in the vocabulary, which disregards not only the order of words but also their adjacency. This means that sentences such as 'they write about machine learning' and 'they are learning to write about a machine' will be virtually indistinguishable, even though the former is about machine learning and the latter is not. It may therefore sometimes be necessary to include phrases consisting of multiple words in the dictionary and treat them as single features. In the information retrieval literature, a multi-word phrase is referred to as an *n-gram* (unigram, bigram, trigram and so on).

Taking this idea one step further, we can construct a new feature from two Boolean or categorical features by forming their Cartesian product. For example, if we have

one feature Shape with values Circle, Triangle and Square, and another feature Colour with values Red, Green and Blue, then their Cartesian product would be the feature (Shape, Colour) with values (Circle, Red), (Circle, Green), (Circle, Blue), (Triangle, Red), and so on. The effect that this would have depends on the model being trained. Constructing Cartesian product features for a naive Bayes classifier means that the two original features are no longer treated as independent, and so this reduces the strong bias that naive Bayes models have. This is not the case for tree models, which can already distinguish between all possible pairs of feature values. On the other hand, a newly introduced Cartesian product feature may incur a high information gain, so it can possibly affect the model learned.

There are many other ways of combining features. For instance, we can take arithmetic or polynomial combinations of quantitative features (we saw examples of this in the use of a ☞*kernel* in Example 1.9 on p.43 and Section 7.5). One attractive possibility is to first apply concept learning or subgroup discovery, and then use these concepts or subgroups as new Boolean features. For instance, in the dolphin domain we could first learn subgroups such as $\mathsf{Length} = [3,5] \wedge \mathsf{Gills} = \mathsf{no}$ and use these as Boolean features in a subsequent tree model. Notice that this expands the expressive power of tree models through the use of negation: e.g., $(\mathsf{Length} = [3,5] \wedge \mathsf{Gills} = \mathsf{no}) = \mathsf{false}$ is equivalent to the disjunction $\mathsf{Length} \neq [3,5] \vee \mathsf{Gills} = \mathsf{yes}$, which is not directly expressible by a feature tree.

Once we have constructed new features it is often a good idea to select a suitable subset of them prior to learning. Not only will this speed up learning as fewer candidate features need to be considered, it also helps to guard against overfitting. There are two main approaches to feature selection. The *filter* approach scores features on a particular metric and the top-scoring features are selected. Many of the metrics we have seen so far can be used for feature scoring, including information gain, the $\chi^2$ statistic, the correlation coefficient, to name just a few. An interesting variation is provided by the *Relief* feature selection method, which repeatedly samples a random instance $x$ and finds its nearest hit $h$ (instance of the same class) as well as its nearest miss $m$ (instance of opposite class). The $i$-th feature's score is then decreased by $\mathrm{Dis}(x_i, h_i)^2$ and increased by $\mathrm{Dis}(x_i, m_i)^2$, where Dis is some distance measure (e.g., Euclidean distance for quantitative features, Hamming distance for categorical features). The intuition is that we want to move closer to the nearest hit while differentiating from the nearest miss.

One drawback of a simple filter approach is that no account is taken of redundancy between features. Imagine, for the sake of the argument, duplicating a promising feature in the data set: both copies score equally high and will be selected, whereas the second one provides no added value in the context of the first one. Secondly, feature filters do not detect dependencies between features as they are solely based on marginal

distributions. For example, consider two Boolean features such that half the positives have the value true for both features and the other half have the value false for both, whereas all negatives have opposite values (again distributed half-half over the two possibilities). It follows that each feature in isolation has zero information gain and hence is unlikely to be selected by a feature filter, despite their combination being a perfect classifier. One could say that feature filters are good at picking out possible root features for a decision tree, but not necessarily good at selecting features that are useful further down the tree.

To detect features that are useful in the context of other features we need to evaluate sets of features; this usually goes under the name of *wrapper* approaches. The idea is that feature selection is 'wrapped' in a search procedure that usually involves training and evaluating a model with a candidate set of features. *Forward selection* methods start with an empty set of features and add features to the set one at a time, as long as they improve the performance of the model. *Backward elimination* starts with the full set of features and aims at improving performance by removing features one at a time. Since there are an exponential number of subsets of features it is usually not feasible to search all possible subsets, and most approaches apply a 'greedy' search algorithm that never reconsiders the choices it makes.

### Matrix transformations and decompositions

We can also view feature construction and selection from a geometric perspective, assuming quantitative features. To this end we represent our data set as a matrix $\mathbf{X}$ with $n$ data points in rows and $d$ features in columns, which we want to transform into a new matrix $\mathbf{W}$ with $n$ rows and $r$ columns by means of matrix operations. To simplify matters a bit, we assume that $\mathbf{X}$ is zero-centred and that $\mathbf{W} = \mathbf{XT}$ for some $d$-by-$r$ transformation matrix $\mathbf{T}$. For example, feature scaling corresponds to $\mathbf{T}$ being a $d$-by-$d$ diagonal matrix; this can be combined with feature selection by removing some of $\mathbf{T}$'s columns. A rotation is achieved by $\mathbf{T}$ being orthogonal, i.e., $\mathbf{TT}^{\mathrm{T}} = \mathbf{I}$. Clearly, several such transformations can be combined (see also Background 1.2 on p.24).

One of the best-known algebraic feature construction methods is *principal component analysis* (*PCA*). Principal components are new features constructed as linear combinations of the given features. The first principal component is given by the direction of maximum variance in the data; the second principal component is the direction of maximum variance orthogonal to the first component, and so on. PCA can be explained in a number of different ways: here, we will derive it by means of the *singular value decomposition* (*SVD*). Any $n$-by-$d$ matrix can be uniquely written as a product of three matrices with special properties:

$$\mathbf{X} = \mathbf{U\Sigma V}^{\mathrm{T}} \tag{10.2}$$

Here, $\mathbf{U}$ is an $n$-by-$r$ matrix, $\mathbf{\Sigma}$ is an $r$-by-$r$ matrix and $\mathbf{V}$ is an $d$-by-$r$ matrix (for the moment we will assume $r = d < n$). Furthermore, $\mathbf{U}$ and $\mathbf{V}$ are orthogonal (hence rotations) and $\mathbf{\Sigma}$ is diagonal (hence a scaling). The columns of $\mathbf{U}$ and $\mathbf{V}$ are known as the left and right singular vectors, respectively; and the values on the diagonal of $\mathbf{\Sigma}$ are the corresponding singular values. It is customary to order the columns of $\mathbf{V}$ and $\mathbf{U}$ so that the singular values are decreasing from top-left to bottom-right.

Now, consider the $n$-by-$r$ matrix $\mathbf{W} = \mathbf{U\Sigma}$, and notice that $\mathbf{XV} = \mathbf{U\Sigma V}^{\mathrm{T}}\mathbf{V} = \mathbf{U\Sigma} = \mathbf{W}$ by the orthogonality of $\mathbf{V}$. In other words, we can construct $\mathbf{W}$ from $\mathbf{X}$ by means of the transformation $\mathbf{V}$: this is the reformulation of $\mathbf{X}$ in terms of its principal components. The newly constructed features are found in $\mathbf{U\Sigma}$: the first row is the first principal component, the second row is the second principal component, and so on. These principal components have a geometric interpretation as the directions in which $\mathbf{X}$ has largest, second-largest, … variance. Assuming the data is zero-centred, these directions can be brought out by a combination of rotation and scaling, which is exactly what PCA does.

We can also use SVD to rewrite the scatter matrix in a standard form:

$$\mathbf{S} = \mathbf{X}^{\mathrm{T}}\mathbf{X} = \left(\mathbf{U\Sigma V}^{\mathrm{T}}\right)^{\mathrm{T}}\left(\mathbf{U\Sigma V}^{\mathrm{T}}\right) = \left(\mathbf{V\Sigma U}^{\mathrm{T}}\right)\left(\mathbf{U\Sigma V}^{\mathrm{T}}\right) = \mathbf{V\Sigma}^2\mathbf{V}^{\mathrm{T}}$$

This is known as the *eigendecomposition* of the matrix $\mathbf{S}$: the columns of $\mathbf{V}$ are the eigenvectors of $\mathbf{S}$, and the elements on the diagonal of $\mathbf{\Sigma}^2$ – which is itself a diagonal matrix – are the eigenvalues. The right singular vectors of the data matrix $\mathbf{X}$ are the eigenvectors of the scatter matrix $\mathbf{S} = \mathbf{X}^{\mathrm{T}}\mathbf{X}$, and the singular values of $\mathbf{X}$ are the square root of the eigenvalues of $\mathbf{S}$. We can derive a similar expression for the Gram matrix $\mathbf{G} = \mathbf{XX}^{\mathrm{T}} = \mathbf{U\Sigma}^2\mathbf{U}^{\mathrm{T}}$, from which we see that the eigenvectors of the Gram matrix are the left singular vectors of $\mathbf{X}$. This demonstrates that in order to perform principal components analysis it is sufficient to perform an eigendecomposition of the scatter or Gram matrices, rather than a full singular value decomposition.

We have seen something resembling SVD in Section 1.1, where we considered the following matrix product:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The matrix on the left expresses people's preferences for films (in columns). The right-hand side decomposes or factorises this into film genres: the first matrix quantifies people's appreciation of genres; the last matrix associates films with genres; and the middle matrix tells us the weight of each genre in determining preferences. This is

not actually the decomposition computed by SVD, because the left and right matrices in the product are not orthogonal. However, one could argue that this factorisation better captures the data, because the person-by-genre and the film-by-genre matrices are Boolean and sparse, which they won't be in the SVD. The downside is that adding integer or Boolean constraints makes the decomposition problem non-convex (there are local optima) and computationally harder. Matrix decomposition with additional constraints is a very active research area.

These matrix decomposition techniques are often used for dimensionality reduction. The *rank* of an $n$-by-$d$ matrix is $d$ (assuming $d < n$ and no columns are linear combinations of other columns). The above decompositions are full-rank because $r = d$, and hence the data matrix is reconstructed exactly. A low-rank approximation of a matrix is a factorisation where $r$ is chosen as small as possible while still sufficiently approximating the original matrix. The reconstruction error is usually measured as the sum of the squared differences of the entries in $\mathbf{X}$ and the corresponding entries in $\mathbf{U\Sigma V}^{\mathrm{T}}$. It can be shown that a truncated singular value decomposition with $r < d$ results in the lowest reconstruction error in this sense among all decompositions of rank up to $r$. Truncated SVD and PCA are popular ways to combine feature construction and feature selection for quantitative features.

One interesting aspect of matrix decompositions such as SVD is that they expose a previously hidden variable in the data. This can be seen as follows. Consider a decomposition or approximation $\mathbf{U\Sigma V}^{\mathrm{T}}$ with diagonal $\mathbf{\Sigma}$ but not necessarily orthogonal $\mathbf{U}$ and $\mathbf{V}$, and denote the $i$-the column of $\mathbf{U}$ and $\mathbf{V}$ as $\mathbf{U}_{\cdot i}$ (an $n$-vector) and $\mathbf{V}_{\cdot i}$ (a $d$-vector). Then $\mathbf{U}_{\cdot i}\sigma_i(\mathbf{V}_{\cdot i})^{\mathrm{T}}$ is an outer product that produces an $n$-by-$d$ matrix with rank 1 ($\sigma_i$ denotes the $i$-th diagonal value of $\mathbf{\Sigma}$). A rank-1 matrix is such that every column is obtained from a single basis vector multiplied by a scalar (and the same for rows). Assuming $\mathbf{U}$ and $\mathbf{V}$ have rank $r$ these basis vectors are independent and so summing up these rank-1 matrices for all $i$ produces the original matrix:

$$\mathbf{U\Sigma V}^{\mathrm{T}} = \sum_{i=1}^{r} \mathbf{U}_{\cdot i}\sigma_i(\mathbf{V}_{\cdot i})^{\mathrm{T}}$$

For example, the film rating matrix can be written as follows:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 2 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The matrices on the right can be interpreted as rating models conditioned on genre.

Exposing hidden variables in the data is one of the main applications of matrix decomposition methods. For example, in information retrieval PCA is known under the

name *latent semantic indexing* (*LSA*) ('latent' is synonymous with 'hidden'). Instead of film genres, LSA uncovers document topics by decomposing matrices containing word counts per document, under the assumption that the word counts per topic are independent and can thus simply be added up.[3] The other main application of matrix factorisation is *completion* of missing entries in a matrix, the idea being that if we approximate the observed entries in the matrix as closely as possible using a low-rank decomposition, this allows us to infer the missing entries.

## 10.4 Features: Summary and further reading

In this chapter we have given features some long-overdue attention. Features are the telescopes through which we observe the data universe and therefore an important unifying force in machine learning. Features are related to measurements in science, but there is no widespread consensus on how to formalise and categorise different measurements – I have taken inspiration from Stevens' scales of measurements (Stevens, 1946), but otherwise aimed to stay close to current practice in machine learning.

☞ The main kinds of feature distinguished in Section 10.1 are categorical, ordinal and quantitative features. The latter are expressed on a quantitative scale and admit the calculation of the widest range of statistics of tendency (mean, median, mode; see (von Hippel, 2005) for a discussion of rules of thumb regarding these), dispersion (variance and standard deviation, range, interquartile range) and shape (skewness and kurtosis). In machine learning quantitative features are often referred to as continuous features, but I think this term is inappropriate as it wrongly suggests that their defining feature is somehow an unlimited precision. It is important to realise that quantitative features do not necessarily have an additive scale: e.g., quantitative features expressing a probability are expressed on a multiplicative scale, and the use of Euclidean distance, say, would be inappropriate for non-additive features. Ordinal features have order but not scale; and categorical features (also called nominal or discrete) have neither order nor scale.

☞ Structured features are first-order logical statements that refer to parts of objects by means of local variables and use some kind of aggregation, such as existential quantification or counting, to extract a property of the main object. Constructing first-order features prior to learning is often referred to as propositionalisation;

[3] Other models are possible: e.g., in Boolean matrix decomposition the matrix product is changed to a Boolean product in which integer addition is replaced by Boolean disjunction (so that $1+1=1$), with the effect that additional topics do not provide additional explanatory power for the occurrence of a word in a document.

Kramer *et al.* (2000) and Lachiche (2010) give surveys, and an experimental comparison of different approaches is carried out by Krogel *et al.* (2003).

☞ In Section 10.2 we looked at a number of feature transformations. Discretisation and thresholding are the best-known of these, turning a quantitative feature into a categorical or a Boolean one. One of the most effective discretisation methods is the recursive partitioning algorithm using information gain to find the thresholds and a stopping criterion derived from the minimum description length principle proposed by Fayyad and Irani (1993). Other overviews and proposals are given by Boullé (2004, 2006). The agglomerative merging approach using $\chi^2$ was proposed by Kerber (1992).

☞ We have seen that in a two-class setting, supervised discretisation can be visualised by means of coverage curves. This then naturally leads to the idea of using these coverage curves and their convex hull to calibrate rather than just discretise the features. After all, ordinal and quantitative features are univariate rankers and scoring classifiers and thus the same classifier calibration methods can be applied, in particular logistic and isotonic calibration as discussed in Section 7.4. The calibrated features live in probability space, but we might prefer to work with log-odds space instead as this is additive rather than multiplicative. Fitting data to a fixed linear decision boundary in calibrated log-odds space is closely related to training a naive Bayes model. Isotonic calibration leads to piecewise axis-parallel decision boundaries; owing to the discretising nature of isotonic calibration this can be understood as the constructing of a grouping model, even if the original model in the uncalibrated space was a grading model.

☞ Section 10.3 was devoted to feature construction and selection. Early approaches to feature construction and constructive induction were proposed by Ragavan and Rendell (1993); Donoho and Rendell (1995). The instance-based Relief feature selection method is due to Kira and Rendell (1992) and extended by Robnik-Sikonja and Kononenko (2003). The distinction between filter approaches to feature selection – which evaluate features on their individual merits – and wrapper approaches, which evaluate sets of features, is originally due to Kohavi and John (1997). Hall (1999) proposes a filter approach called correlation-based feature selection that aims at combining the best of both worlds. Guyon and Elisseeff (2003) give an excellent introduction to feature selection.

☞ Finally, we looked at feature construction and selection from a linear algebra perspective. Matrix decomposition and factorisation is an actively researched technique that was instrumental in winning a recent film recommendation challenge worth $1 million (Koren *et al.*, 2009). Decomposition techniques employing additional constraints include non-negative matrix decomposition (Lee *et al.*, 1999).

Boolean matrix decomposition is studied by Miettinen (2009). Mahoney and Drineas (2009) describe a matrix decomposition technique that uses actual columns and rows of the data matrix to preserve sparsity (unlike SVD which produces dense matrices even if the original matrix is sparse). Latent semantic indexing and a probabilistic extension is described by Hofmann (1999). Ding and He (2004) discuss the relationship between $K$-means clustering and principal component analysis.

CHAPTER 7

# Linear models

AFTER DEALING WITH logical models in the preceding chapters we now move on to a quite different kind of model. The models in this chapter and the next are defined in terms of the geometry of instance space. Geometric models most often assume that instances are described by $d$ real-valued features, and thus $\mathscr{X} = \mathbb{R}^d$. For example, we could describe objects by their position on a map in terms of longitude and latitude ($d = 2$), or in the real world by longitude, latitude and altitude ($d = 3$). While most real-valued features are not intrinsically geometric – think of a person's age or an object's temperature – we can still imagine them being plotted in a $d$-dimensional Cartesian coordinate system. We can then use geometric concepts such as lines and planes to impose structure on this space, for instance in order to build a classification model. Alternatively, we can use the geometric notion of distance to represent similarity, on the basis that if two points are close together they have similar feature values and thus can be expected to behave similarly with respect to the property of interest. Such distance-based models are the subject of the next chapter. In this chapter we will look at models that can be understood in terms of lines and planes, commonly called *linear models*.

Linearity plays a fundamental role in mathematics and related disciplines, and the mathematics of linear models is well-understood (see Background 7.1 for the most important concepts). In machine learning, linear models are of particular interest because of their simplicity (remember our rule of thumb 'everything should be made as

If $x_1$ and $x_2$ are two scalars or vectors of the same dimension and $\alpha$ and $\beta$ are arbitrary scalars, then $\alpha x_1 + \beta x_2$ is called a *linear combination* of $x_1$ and $x_2$. If $f$ is a *linear function* of $x$, then

$$f(\alpha x_1 + \beta x_2) = \alpha f(x_1) + \beta f(x_2)$$

In words, the function value of a linear combination of some inputs is a linear combination of their function values. As a special case, if $\beta = 1 - \alpha$ we are taking a weighted average of $x_1$ and $x_2$, and the linearity of $f$ then means that the function value of the weighted average is the weighted average of the function values.

Linear functions take particular forms, depending on the domain and codomain of $f$. If $x$ and $f(x)$ are scalars, it follows that $f$ is of the form $f(x) = a + bx$ for some constants $a$ and $b$; $a$ is called the *intercept* and $b$ the *slope*. If $\mathbf{x} = (x_1, \ldots, x_d)$ is a vector and $f(\mathbf{x})$ is a scalar, then $f$ is of the form

$$f(\mathbf{x}) = a + b_1 x_1 + \ldots + b_d x_d = a + \mathbf{b} \cdot \mathbf{x} \tag{7.1}$$

with $\mathbf{b} = (b_1, \ldots, b_d)$. The equation $f(\mathbf{x}) = 0$ defines a plane in $\mathbb{R}^d$ perpendicular to the *normal vector* $\mathbf{b}$.

The most general case is where $f(\mathbf{x})$ is a $d'$-dimensional vector, in which case $f$ is of the form $f(\mathbf{x}) = \mathbf{M}\mathbf{x} + \mathbf{t}$, where $\mathbf{M}$ is a $d'$-by-$d$ matrix representing a *linear transformation* such as a rotation or a scaling, and $\mathbf{t}$ is a $d'$-vector representing a translation. In this case $f$ is called an *affine transformation* (the difference between linear and affine transformations is that the former maps the origin to itself; notice that a linear function of the form of Equation 7.1 is a linear transformation only if the intercept is 0).

In all these forms we can avoid representing the intercept $a$ or the translation $\mathbf{t}$ separately by using homogeneous coordinates. For instance, by writing $\mathbf{b}^\circ = (a, b_1, \ldots, b_d)$ and $\mathbf{x}^\circ = (1, x_1, \ldots, x_d)$ in Equation 7.1 we have $f(\mathbf{x}) = \mathbf{b}^\circ \cdot \mathbf{x}^\circ$ (see also Background 1.2 on p.24).

Examples of non-linear functions are the polynomials in $x$ of degree $p > 1$: $g(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_p x^p = \sum_{i=0}^{p} a_i x^i$. Other non-linear functions can be approximated by a polynomial through their Taylor expansion. The *linear approximation* of a function $g$ at $x_0$ is $g(x_0) + g'(x_0)(x - x_0)$, where $g'(x)$ is the derivative of $x$. A *piecewise linear* approximation is obtained by combining several linear approximations at different points $x_0$.

**Background 7.1.** Linear models.

simple as possible, but not simpler' that we introduced on p.30). Here are a couple of manifestations of this simplicity.

☞ Linear models are *parametric*, meaning that they have a fixed form with a small number of numeric parameters that need to be learned from data. This is

different from tree or rule models, where the structure of the model (e.g., which features to use in the tree, and where) is not fixed in advance.

☞ Linear models are stable, which is to say that small variations in the training data have only limited impact on the learned model. Tree models tend to vary more with the training data, as the choice of a different split at the root of the tree typically means that the rest of the tree is different as well.

☞ Linear models are less likely to overfit the training data than some other models, largely because they have relatively few parameters. The flipside of this is that they sometimes lead to *underfitting*: e.g., imagine you are learning where the border runs between two countries from labelled samples, then a linear model is unlikely to give a good approximation.

The last two points can be summarised by saying that linear models have low variance but high bias. Such models are often preferable when you have limited data and want to avoid overfitting. High variance–low bias models such as decision trees are preferable if data is abundant but underfitting is a concern. It is usually a good idea to start with simple, high-bias models such as linear models and only move on to more elaborate models if the simpler ones appear to be underfitting.

Linear models exist for all predictive tasks, including classification, probability estimation and regression. Linear regression, in particular, is a well-studied problem that can be solved by the least-squares method, which is the topic of the next section. We will look at a number of other linear models in this chapter, including least-squares classification (also in Section 7.1), the perceptron in Section 7.2, and the support vector machine in Section 7.3. We will also find out how these models can be turned into probability estimators in Section 7.4. Finally, Section 7.5 briefly discusses how each of these methods could learn non-linear models by means of so-called kernel functions.

## 7.1 The least-squares method

We start by introducing a method that can be used to learn linear models for classification and regression. Recall that the regression problem is to learn a function estimator $\hat{f} : \mathscr{X} \to \mathbb{R}$ from examples $(x_i, f(x_i))$, where in this chapter we assume $\mathscr{X} = \mathbb{R}^d$. The differences between the actual and estimated function values on the training examples are called *residuals* $\epsilon_i = f(x_i) - \hat{f}(x_i)$. The *least-squares method*, introduced by Carl Friedrich Gauss in the late eighteenth century, consists in finding $\hat{f}$ such that $\sum_{i=1}^{n} \epsilon_i^2$ is minimised. The following example illustrates the method in the simple case of a single feature, which is called *univariate regression*.

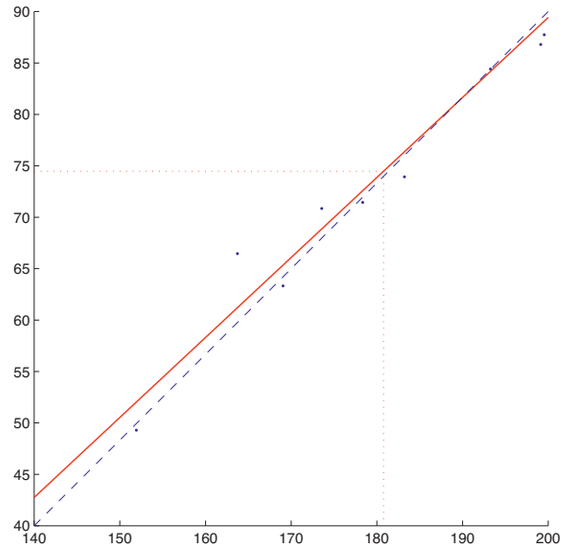



**Figure 7.1.** The red solid line indicates the result of applying linear regression to 10 measurements of body weight (on the $y$-axis, in kilograms) against body height (on the $x$-axis, in centimetres). The orange dotted lines indicate the average height $\overline{h} = 181$ and the average weight $\overline{w} = 74.5$; the regression coefficient $\hat{b} = 0.78$. The measurements were simulated by adding normally distributed noise with mean 0 and variance 5 to the true model indicated by the blue dashed line ($b = 0.83$).

---

**Example 7.1 (Univariate linear regression).** Suppose we want to investigate the relationship between people's height and weight. We collect $n$ height and weight measurements $(h_i, w_i), 1 \leq i \leq n$. Univariate linear regression assumes a linear equation $w = a + bh$, with parameters $a$ and $b$ chosen such that the sum of squared residuals $\sum_{i=1}^{n}(w_i - (a + bh_i))^2$ is minimised. In order to find the parameters we take partial derivatives of this expression, set the partial derivatives to 0 and solve for $a$ and $b$:

$$\frac{\partial}{\partial a}\sum_{i=1}^{n}(w_i - (a + bh_i))^2 = -2\sum_{i=1}^{n}(w_i - (a + bh_i)) = 0 \qquad \Rightarrow \hat{a} = \overline{w} - \hat{b}\overline{h}$$

$$\frac{\partial}{\partial b}\sum_{i=1}^{n}(w_i - (a + bh_i))^2 = -2\sum_{i=1}^{n}(w_i - (a + bh_i))h_i = 0$$

$$\Rightarrow \hat{b} = \frac{\sum_{i=1}^{n}(h_i - \overline{h})(w_i - \overline{w})}{\sum_{i=1}^{n}(h_i - \overline{h})^2}$$

So the solution found by linear regression is $w = \hat{a} + \hat{b}h = \overline{w} + \hat{b}(h - \overline{h})$; see Figure 7.1 for an example.

---

It is worthwhile to note that the expression for the *regression coefficient* or slope $\hat{b}$ derived in this example has $n$ times the covariance between $h$ and $w$ in the enumerator and $n$ times the variance of $h$ in the denominator. This is true in general: for a feature $x$ and a target variable $y$, the regression coefficient is

$$\hat{b} = n\frac{\sigma_{xy}}{n\sigma_{xx}} = \frac{\sigma_{xy}}{\sigma_{xx}}$$

(Here I use $\sigma_{xx}$ as an alternative notation for $\sigma_x^2$, the variance of variable $x$.) This can be understood by noting that the covariance is measured in units of $x$ times units of $y$ (e.g., metres times kilograms in Example 7.1) and the variance in units of $x$ squared (e.g., metres squared), so their quotient is measured in units of $y$ per unit of $x$ (e.g., kilograms per metre).

We can notice a few more useful things. The intercept $\hat{a}$ is such that the regression line goes through $(\overline{x}, \overline{y})$. Adding a constant to all $x$-values (a translation) will affect only the intercept but not the regression coefficient (since it is defined in terms of deviations from the mean, which are unaffected by a translation). So we could *zero-centre* the $x$-values by subtracting $\overline{x}$, in which case the intercept is equal to $\overline{y}$. We could even subtract $\overline{y}$ from all $y$-values to achieve a zero intercept, without changing the problem in an essential way.

Furthermore, suppose we replace $x_i$ with $x'_i = x_i/\sigma_{xx}$ and likewise $\overline{x}$ with $\overline{x'} = \overline{x}/\sigma_{xx}$, then we have that $\hat{b} = \frac{1}{n}\sum_{i=1}^{n}(x'_i - \overline{x'})(y_i - \overline{y}) = \sigma_{x'y}$. In other words, if we *normalise* $x$ by dividing all its values by $x$'s variance, we can take the covariance between the normalised feature and the target variable as regression coefficient. In other words, univariate linear regression can be understood as consisting of two steps:

1. normalisation of the feature by dividing its values by the feature's variance;

2. calculating the covariance of the target variable and the normalised feature.

We will see below how these two steps change when dealing with more than one feature.

Another important point to note is that the sum of the residuals of the least-squares solution is zero:

$$\sum_{i=1}^{n}(y_i - (\hat{a} + \hat{b}x_i)) = n(\overline{y} - \hat{a} - \hat{b}\overline{x}) = 0$$

The result follows because $\hat{a} = \overline{y} - \hat{b}\overline{x}$, as derived in Example 7.1. While this property is intuitively appealing, it is worth keeping in mind that it also makes linear regression susceptible to *outliers*: points that are far removed from the regression line, often because of measurement errors.

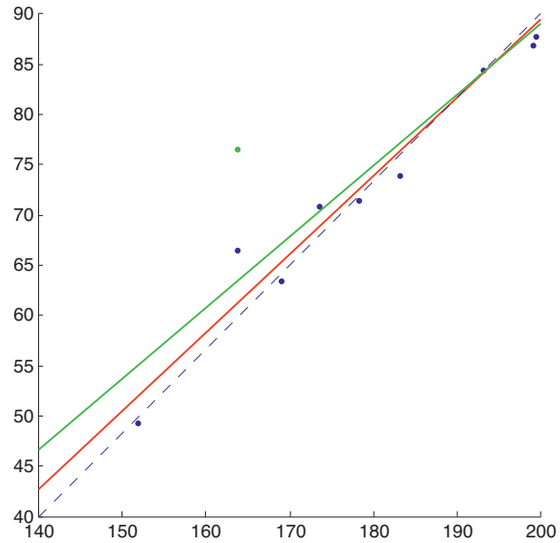



**Figure 7.2.** The effect of an outlier in univariate regression. One of the blue points got moved up 10 units to the green point, changing the red regression line to the green line.

---

**Example 7.2 (The effect of outliers).** Suppose that, as the result of a transcription error, one of the weight values in Figure 7.1 is increased by 10 kg. Figure 7.2 shows that this has a considerable effect on the least-squares regression line.

---

Despite this sensitivity to outliers, the least-squares method usually works surprisingly well for such a simple method. How can it be justified? One way to look at this is to assume that the true function $f$ is indeed linear, but that the observed $y$-values are contaminated with random noise. That is, our examples are $(x_i, f(x_i) + \epsilon_i)$ rather than $(x_i, f(x_i))$, and we assume that $f(x) = ax + b$ for some $a$ and $b$. If we knew $a$ and $b$ we could work out exactly what the residuals are, and if we knew $\sigma^2$ we could calculate the probability of observing that set of residuals. Since we don't know $a$ and $b$ we have to estimate them, and the estimate we want is the value of $a$ and $b$ that maximises the probability of the residuals. We will see in Chapter 9 that this so-called ☞*maximum-likelihood estimate* is exactly the least-squares solution.

Variants of the least-squares method exist. Here we discussed *ordinary* least squares, which assumes that only the $y$-values are contaminated with random noise. *Total* least squares generalises this to the situation that both $x$- and $y$-values are noisy, but this does not necessarily have a unique solution.

---

$\mathbf{X}$ usually denotes an $n$-by-$d$ data matrix containing $n$ instances in rows described by $d$ features or variables in columns. $\mathbf{X}_{r\cdot}$ denotes the $r$-th row of $\mathbf{X}$, $\mathbf{X}_{\cdot c}$ denotes the $c$-th column, and $\mathbf{X}_{rc}$ denotes the entry in the $r$-th row and $c$-th column. We also use $i$ and $j$ to range over rows and columns, respectively. The $j$-th column mean is defined as $\mu_j = \frac{1}{n}\sum_{i=1}^{n}\mathbf{X}_{ij}$; $\boldsymbol{\mu}^{\mathrm{T}}$ is a row vector containing all column means. If $\mathbf{1}$ is an $n$-vector containing only ones, then $\mathbf{1}\boldsymbol{\mu}^{\mathrm{T}}$ is an $n$-by-$d$ matrix whose rows are $\boldsymbol{\mu}^{\mathrm{T}}$; hence $\mathbf{X}' = \mathbf{X} - \mathbf{1}\boldsymbol{\mu}^{\mathrm{T}}$ has mean zero in each column and is referred to as the *zero-centred* data matrix.
The *scatter matrix* is the $d$-by-$d$ matrix $\mathbf{S} = \mathbf{X}'^{\mathrm{T}}\mathbf{X}' = \left(\mathbf{X} - \mathbf{1}\boldsymbol{\mu}^{\mathrm{T}}\right)^{\mathrm{T}}\left(\mathbf{X} - \mathbf{1}\boldsymbol{\mu}^{\mathrm{T}}\right) = \mathbf{X}^{\mathrm{T}}\mathbf{X} - n\mathbf{M}$, where $\mathbf{M} = \boldsymbol{\mu}\boldsymbol{\mu}^{\mathrm{T}}$ is a $d$-by-$d$ matrix whose entries are products of column means $\mathbf{M}_{jc} = \mu_j\mu_c$. The *covariance matrix* of $\mathbf{X}$ is $\boldsymbol{\Sigma} = \frac{1}{n}\mathbf{S}$ whose entries are the pairwise covariances $\sigma_{jc} = \frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{X}_{ij} - \mu_j\right)\left(\mathbf{X}_{ic} - \mu_c\right) = \frac{1}{n}\left(\sum_{i=1}^{n}\mathbf{X}_{ij}\mathbf{X}_{ic} - \mu_i\mu_c\right)$. Two uncorrelated features have a covariance close to 0; positively correlated features have a positive covariance, indicating a certain tendency to increase or decrease together; a negative covariance indicates that if one feature increases, the other tends to decrease and vice versa. $\sigma_{jj} = \frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{X}_{ij} - \mu_j\right)^2 = \frac{1}{n}\left(\sum_{i=1}^{n}\mathbf{X}_{ij}^2 - \mu_j^2\right)$ is the *variance* of column $j$, also denoted as $\sigma_j^2$. The variance is always positive and indicates the spread of the values of a feature around their mean.
A small example clarifies these definitions:

$$\mathbf{X} = \begin{pmatrix} 5 & 0 \\ 3 & 5 \\ 1 & 7 \end{pmatrix} \quad \mathbf{1}\boldsymbol{\mu}^{\mathrm{T}} = \begin{pmatrix} 3 & 4 \\ 3 & 4 \\ 3 & 4 \end{pmatrix} \quad \mathbf{X}' = \begin{pmatrix} 2 & -4 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \quad \mathbf{G} = \begin{pmatrix} 25 & 15 & 5 \\ 15 & 34 & 38 \\ 5 & 38 & 50 \end{pmatrix}$$

$$\mathbf{X}^{\mathrm{T}}\mathbf{X} = \begin{pmatrix} 35 & 22 \\ 22 & 74 \end{pmatrix} \quad \mathbf{M} = \begin{pmatrix} 9 & 12 \\ 12 & 16 \end{pmatrix} \quad \mathbf{S} = \begin{pmatrix} 8 & -14 \\ -14 & 26 \end{pmatrix} \quad \boldsymbol{\Sigma} = \begin{pmatrix} 8/3 & -14/3 \\ -14/3 & 26/3 \end{pmatrix}$$

We see that the two features are negatively correlated and that the second feature has the larger variance. Another way to calculate the scatter matrix is as a sum of outer products, one for each data point: $\mathbf{S} = \sum_{i=1}^{n}\left(\mathbf{X}_{i\cdot} - \boldsymbol{\mu}^{\mathrm{T}}\right)^{\mathrm{T}}\left(\mathbf{X}_{i\cdot} - \boldsymbol{\mu}^{\mathrm{T}}\right)$. In our example we have

$$\left(\mathbf{X}_{1\cdot} - \boldsymbol{\mu}^{\mathrm{T}}\right)^{\mathrm{T}}\left(\mathbf{X}_{1\cdot} - \boldsymbol{\mu}^{\mathrm{T}}\right) = \begin{pmatrix} 2 \\ -4 \end{pmatrix}\begin{pmatrix} 2 & -4 \end{pmatrix} = \begin{pmatrix} 4 & -8 \\ -8 & 16 \end{pmatrix}$$

$$\left(\mathbf{X}_{2\cdot} - \boldsymbol{\mu}^{\mathrm{T}}\right)^{\mathrm{T}}\left(\mathbf{X}_{2\cdot} - \boldsymbol{\mu}^{\mathrm{T}}\right) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\left(\mathbf{X}_{3\cdot} - \boldsymbol{\mu}^{\mathrm{T}}\right)^{\mathrm{T}}\left(\mathbf{X}_{3\cdot} - \boldsymbol{\mu}^{\mathrm{T}}\right) = \begin{pmatrix} -2 \\ 3 \end{pmatrix}\begin{pmatrix} -2 & 3 \end{pmatrix} = \begin{pmatrix} 4 & -6 \\ -6 & 9 \end{pmatrix}$$

---

**Background 7.2.** Some more matrix notation.

### Multivariate linear regression

In order to deal with an arbitrary number of features it will be useful to employ matrix notation (see Background 7.2). We can write univariate linear regression in matrix form as

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} a + \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} b + \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

$$\mathbf{y} = \mathbf{a} + \mathbf{X}\mathbf{b} + \boldsymbol{\epsilon}$$

In the second form of this equation, $\mathbf{y}$, $\mathbf{a}$, $\mathbf{X}$ and $\boldsymbol{\epsilon}$ are $n$-vectors, and $\mathbf{b}$ is a scalar. In case of $d$ features, all that changes is that $\mathbf{X}$ becomes an $n$-by-$d$ matrix, and $\mathbf{b}$ becomes a $d$-vector of regression coefficients.

We can apply the by now familiar trick of using homogeneous coordinates to simplify these equations as follows:

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

$$\mathbf{y} = \mathbf{X}^\circ \mathbf{w} + \boldsymbol{\epsilon}$$

with $\mathbf{X}^\circ$ an $n$-by-$(d+1)$ matrix whose first column is all 1s and the remaining columns are the columns of $\mathbf{X}$, and $\mathbf{w}$ has the intercept as its first entry and the regression coefficients as the remaining $d$ entries. For convenience we will often blur the distinction between these two formulations and state the regression equation as $\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}$ with $\mathbf{X}$ having $d$ columns and $\mathbf{w}$ having $d$ rows – from the context it will be clear whether we are representing the intercept by means of homogeneous coordinates, or have rather zero-centred the target and features to achieve a zero intercept.

In the univariate case we were able to obtain a closed-form solution for $\mathbf{w}$: can we do the same in the multivariate case? First, we are likely to need the covariances between every feature and the target variable. Consider the expression $\mathbf{X}^\mathrm{T}\mathbf{y}$, which is an $n$-vector, the $j$-th entry of which is the product of the $j$-th row of $\mathbf{X}^\mathrm{T}$ – i.e., the $j$-th column of $\mathbf{X}$, which is $(x_{1j}, \ldots, x_{nj})$ – with $(y_1, \ldots, y_n)$:

$$(\mathbf{X}^\mathrm{T}\mathbf{y})_j = \sum_{i=1}^{n} x_{ij} y_i = \sum_{i=1}^{n} (x_{ij} - \mu_j)(y_i - \overline{y}) + n\mu_j \overline{y} = n(\sigma_{jy} + \mu_j \overline{y})$$

Assuming for the moment that every feature is zero-centred, we have $\mu_j = 0$ and thus $\mathbf{X}^\mathrm{T}\mathbf{y}$ is an $n$-vector holding all the required covariances (times $n$).

In the univariate case we needed to normalise the features to have unit variance. In the multivariate case we can achieve this by means of a $d$-by-$d$ scaling matrix: a

diagonal matrix with diagonal entries $1/n\sigma_{jj}$. If $\mathbf{S}$ is a diagonal matrix with diagonal entries $n\sigma_{jj}$, we can get the required scaling matrix by simply inverting $\mathbf{S}$. So our first stab at a solution for the *multivariate regression* problem is

$$\hat{\mathbf{w}} = \mathbf{S}^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} \tag{7.2}$$

As it turns out, the general case requires a more elaborate matrix instead of $\mathbf{S}$:

$$\hat{\mathbf{w}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} \tag{7.3}$$

Let us try to understand the term $(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}$ a bit better. Assume that the features are uncorrelated (meaning the covariance between every pair of different features is 0) in addition to being zero-centred. In the notation of Background 7.2, the covariance matrix $\mathbf{\Sigma}$ is diagonal with entries $\sigma_{jj}$. Since $\mathbf{X}^{\mathrm{T}}\mathbf{X} = n(\mathbf{\Sigma} + \mathbf{M})$, and since the entries of $\mathbf{M}$ are 0 because the columns of $\mathbf{X}$ are zero-centred, this matrix is also diagonal with entries $n\sigma_{jj}$ – in fact, it is the matrix $\mathbf{S}$ referred to above. In other words, assuming zero-centred and uncorrelated features, $(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}$ reduces to our scaling matrix $\mathbf{S}^{-1}$. In the general case we cannot make any assumptions about the features, and $(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}$ *acts as a transformation that decorrelates, centres and normalises the features*.

To make this a bit more concrete, the next example shows how this works out in the bivariate case.

---

**Example 7.3 (Bivariate linear regression in matrix notation).** First, we derive the basic expressions.

$$\begin{aligned}
\mathbf{X}^{\mathrm{T}}\mathbf{X} &= \begin{pmatrix} x_{11} & \cdots & x_{n1} \\ x_{12} & \cdots & x_{n2} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} \\ \vdots & \vdots \\ x_{n1} & x_{n2} \end{pmatrix} = n \begin{pmatrix} \sigma_{11} + \overline{x_1}^2 & \sigma_{12} + \overline{x_1}\,\overline{x_2} \\ \sigma_{12} + \overline{x_1}\,\overline{x_2} & \sigma_{22} + \overline{x_2}^2 \end{pmatrix} \\
(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1} &= \frac{1}{nD} \begin{pmatrix} \sigma_{22} + \overline{x_2}^2 & -\sigma_{12} - \overline{x_1}\,\overline{x_2} \\ -\sigma_{12} - \overline{x_1}\,\overline{x_2} & \sigma_{11} + \overline{x_1}^2 \end{pmatrix} \\
D &= (\sigma_{11} + \overline{x_1}^2)(\sigma_{22} + \overline{x_2}^2) - (\sigma_{12} + \overline{x_1}\,\overline{x_2})^2 \\
\mathbf{X}^{\mathrm{T}}\mathbf{y} &= \begin{pmatrix} x_{11} & \cdots & x_{n1} \\ x_{12} & \cdots & x_{n2} \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = n \begin{pmatrix} \sigma_{1y} + \overline{x_1}\,\overline{y} \\ \sigma_{2y} + \overline{x_2}\,\overline{y} \end{pmatrix}
\end{aligned}$$

We now consider two special cases. The first is that $\mathbf{X}$ is in homogeneous coordinates, i.e., we are really dealing with a univariate problem. In that case we have

$x_{i1} = 1$ for $1 \leq i \leq n$; $\overline{x_1} = 1$; and $\sigma_{11} = \sigma_{12} = \sigma_{1y} = 0$. We then obtain (we write $x$ instead of $x_2$, $\sigma_{xx}$ instead of $\sigma_{22}$ and $\sigma_{xy}$ instead of $\sigma_{2y}$):

$$
\begin{aligned}
(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1} &= \frac{1}{n\sigma_{xx}} \begin{pmatrix} \sigma_{xx} + \overline{x}^2 & -\overline{x} \\ -\overline{x} & 1 \end{pmatrix} \\
\mathbf{X}^{\mathrm{T}}\mathbf{y} &= n \begin{pmatrix} \overline{y} \\ \sigma_{xy} + \overline{x}\,\overline{y} \end{pmatrix} \\
\hat{\mathbf{w}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} &= \frac{1}{\sigma_{xx}} \begin{pmatrix} \sigma_{xx}\overline{y} - \sigma_{xy}\overline{x} \\ \sigma_{xy} \end{pmatrix}
\end{aligned}
$$

This is the same result as obtained in Example 7.1.

The second special case we consider is where we assume $x_1$, $x_2$ and $y$ to be zero-centred, which means that the intercept is zero and $\mathbf{w}$ contains the two regression coefficients. In this case we obtain

$$
\begin{aligned}
(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1} &= \frac{1}{n(\sigma_{11}\sigma_{22} - \sigma_{12}^2)} \begin{pmatrix} \sigma_{22} & -\sigma_{12} \\ -\sigma_{12} & \sigma_{11} \end{pmatrix} \\
\mathbf{X}^{\mathrm{T}}\mathbf{y} &= n \begin{pmatrix} \sigma_{1y} \\ \sigma_{2y} \end{pmatrix} \\
\hat{\mathbf{w}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} &= \frac{1}{(\sigma_{11}\sigma_{22} - \sigma_{12}^2)} \begin{pmatrix} \sigma_{22}\sigma_{1y} - \sigma_{12}\sigma_{2y} \\ \sigma_{11}\sigma_{2y} - \sigma_{12}\sigma_{1y} \end{pmatrix}
\end{aligned}
$$

The last expression shows, e.g., that the regression coefficient for $x_1$ may be non-zero even if $x_1$ doesn't correlate with the target variable ($\sigma_{1y} = 0$), on account of the correlation between $x_1$ and $x_2$ ($\sigma_{12} \neq 0$).

---

Notice that if we do assume $\sigma_{12} = 0$ then the components of $\hat{\mathbf{w}}$ reduce to $\sigma_{jy}/\sigma_{jj}$, which brings us back to Equation 7.2. *Assuming uncorrelated features effectively decomposes a multivariate regression problem into $d$ univariate problems.* We shall see several other examples of decomposing multivariate learning problems into univariate problems in this book – in fact, we have already seen an example in the form of the ☞*naive Bayes* classifier in Chapter 1. So, you may wonder, why take feature correlation into account at all?

The answer is that ignoring feature correlation can be harmful in certain situations. Consider Figure 7.3: on the left, there is little correlation among the features, and as a result the samples provide a lot of information about the true function. On

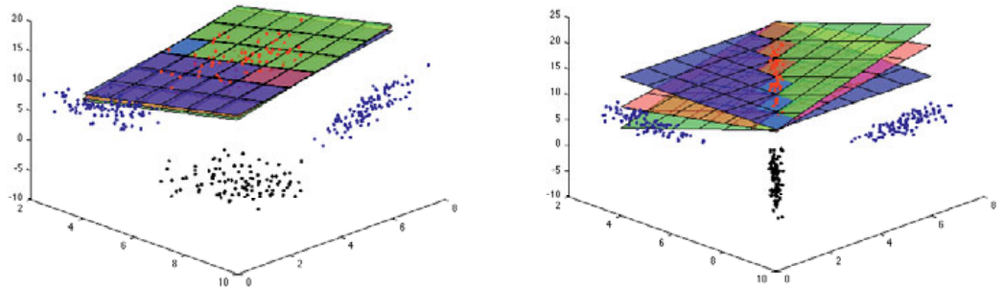


**Figure 7.3.** **(left)** Regression functions learned by linear regression. The true function is $y = x_1 + x_2$ (red plane). The red points are noisy samples of this function; the black points show them projected onto the $(x_1, x_2)$-plane. The green plane indicates the function learned by linear regression; the blue plane is the result of decomposing the problem into two univariate regression problems (blue points). Both are good approximations of the true function. **(right)** The same function, but now $x_1$ and $x_2$ are highly (negatively) correlated. The samples now give much less information about the true function: indeed, from the univariate decomposition it appears that the function is constant.

the right, the features are highly negatively correlated in such a way that the sampled values $y = x_1 + x_2 + \epsilon$ appear nearly constant, as any increase in one feature is accompanied by a nearly equal decrease in the other. As a result, decomposing the problem into two univariate regression problems leads to learning a nearly constant function. To be fair, taking the full covariance matrix into account doesn't do so well either in this example. However, although we will not explore the details here, one advantage of the full covariance approach is that it allows us to recognise that we can't place much confidence in our estimates of the regression parameters in this situation. The computational cost of computing the closed-form solution in Equation 7.3 lies in inverting the $d$-by-$d$ matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$, which can be prohibitive in high-dimensional feature spaces.

## Regularised regression

We have just seen a situation in which least-squares regression can become *unstable*: i.e., highly dependent on the training data. Instability is a manifestation of a tendency to overfit. *Regularisation* is a general method to avoid such overfitting by applying additional constraints to the weight vector. A common approach is to make sure the weights are, on average, small in magnitude: this is referred to as *shrinkage*. To show how this can be achieved, we first write down the least-squares regression problem as an optimisation problem:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\arg\min}\ (\mathbf{y} - \mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\mathbf{w})$$

The right-hand side is just a way to write the sum of squared residuals as a dot product. The regularised version of this optimisation is then as follows:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} (\mathbf{y} - \mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda ||\mathbf{w}||^2 \tag{7.4}$$

where $||\mathbf{w}||^2 = \sum_i w_i^2$ is the squared norm of the vector $\mathbf{w}$, or, equivalently, the dot product $\mathbf{w}^{\mathrm{T}}\mathbf{w}$; $\lambda$ is a scalar determining the amount of regularisation. This regularised problem still has a closed-form solution:

$$\hat{\mathbf{w}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + \lambda \mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y} \tag{7.5}$$

where $\mathbf{I}$ denotes the identity matrix with 1s on the diagonal and 0s everywhere else. Comparing this with Equation 7.3 on p.202 we see that regularisation amounts to adding $\lambda$ to the diagonal of $\mathbf{X}^{\mathrm{T}}\mathbf{X}$, a well-known trick to improve the numerical stability of matrix inversion. This form of least-squares regression is known as *ridge regression*.

An interesting alternative form of regularised regression is provided by the *lasso*, which stands for 'least absolute shrinkage and selection operator'. It replaces the ridge regularisation term $\sum_i w_i^2$ with the sum of absolute weights $\sum_i |w_i|$. (Using terminology that will be introduced in Definition 8.2 on p.235: lasso uses $L_1$ regularisation where ridge regression uses the $L_2$ norm.) The result is that some weights are shrunk, but others are set to 0, and so the lasso regression favours *sparse solutions*. It should be added that lasso regression is quite sensitive to the regularisation parameter $\lambda$, which is usually set on hold-out data or in cross-validation. Also, there is no closed-form solution and so some numerical optimisation technique must be applied.

## Using least-squares regression for classification

So far we have used the least-squares method to construct function approximators. Interestingly, we can also use linear regression to learn a binary classifier by encoding the two classes as real numbers. For instance, we can label the *Pos* positive examples with $y^{\oplus} = +1$ and the *Neg* negative examples with $y^{\ominus} = -1$. It then follows that $\mathbf{X}^{\mathrm{T}}\mathbf{y} = Pos\,\boldsymbol{\mu}^{\oplus} - Neg\,\boldsymbol{\mu}^{\ominus}$, where $\boldsymbol{\mu}^{\oplus}$ and $\boldsymbol{\mu}^{\ominus}$ are $d$-vectors containing each feature's mean values for the positive and negative examples, respectively.

---

**Example 7.4 (Univariate least-squares classifier).** In the univariate case we have $\sum_i x_i y_i = Pos\,\mu^{\oplus} - Neg\,\mu^{\ominus}$; we also know (see Example 7.3) that $\sum_i x_i y_i = n(\sigma_{xy} + \overline{x}\,\overline{y})$, and so $\sigma_{xy} = pos\,\mu^{\oplus} - neg\,\mu^{\ominus} - \overline{x}\,\overline{y}$. Since $\overline{x} = pos\,\mu^{\oplus} + neg\,\mu^{\ominus}$ and $\overline{y} = pos - neg$, we can rewrite the covariance between $x$ and $y$ as $\sigma_{xy} =$

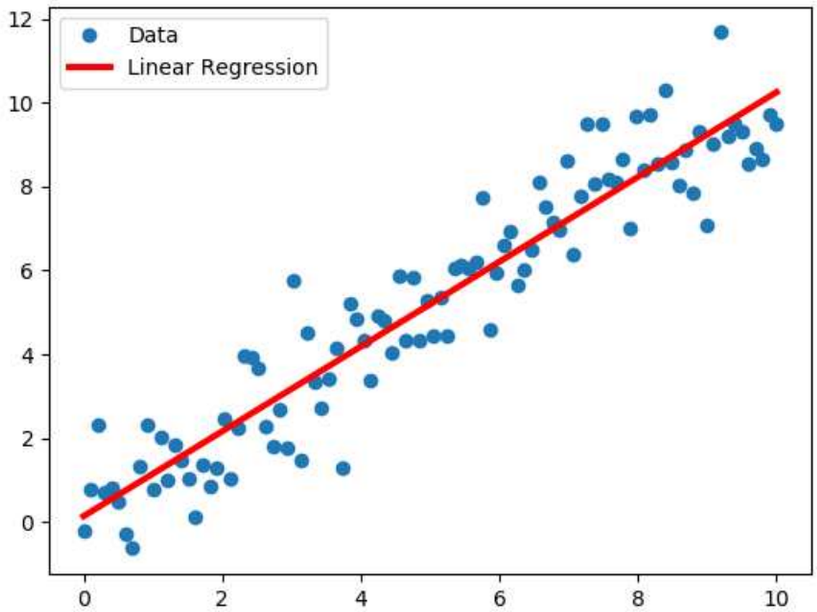
# 6. Gradient Descent

*Linear Regression With Gradient Descent Derivation*

## Pre-Requisites

The only pre-requisites are differentiation and matrix multiplication.

## What is Linear Regression ?



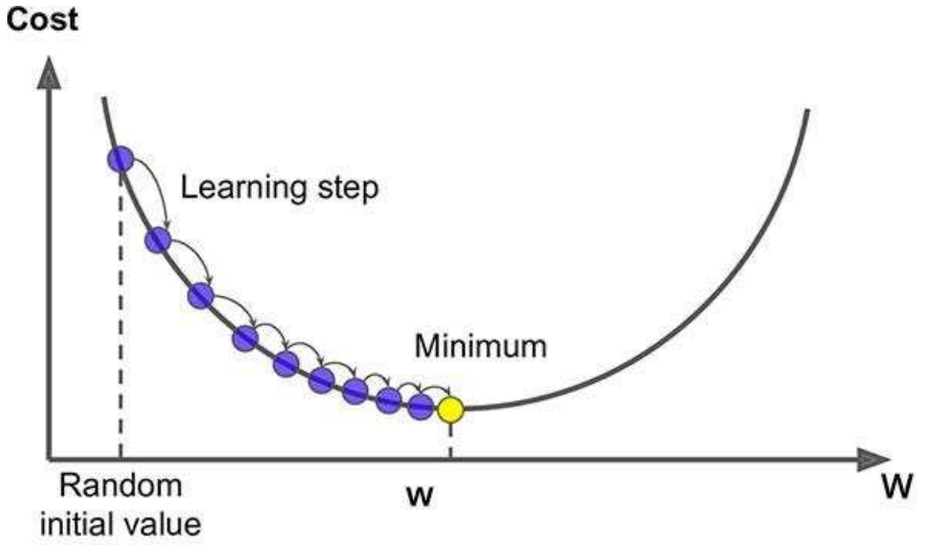

Simple Linear Regression is basically a modelling of linear relationship between linearly dependent variables that can be later used to predict dependent variable values for new independent variables
For this, we use the equation of the line : `y = m * x + c.`

where y is the dependent variable and x is the independent variable

**For example**: We could predict the salary of a person with the years of experience of the person.
Here, Salary is the dependent variable and experience is the independent variable since, we are predicting salary with the help of experience.

**When to use Linear Regression**: Linear regression can be performed on data where there is a good linear relationship between dependent and independent variables.
The degree of linear relationship can be found with help of correlation.

# Gradient Descent




Gradient Descent is a an optimization algorithm that can be used to find the global or local minima of a differentiable function.

## Various Assumptions and definitions before we begin.

Let's say we have an independent variable x and a dependent variable y.
in order to form the relationship between these 2 variables, we have the equation:
`y = x * w + b`
where **w** is weight ( or slope ) ,
**b** is the bias (or intercept ),
**x** is the independent variable column vector(examples),
**y** is the dependent variable column vector(examples)
Our main goal is to find the **w** and **b** that defines the relationship between variable **x** and **y** correctly. We accomplish this with the help of something called as the **Loss Function.**

**Loss Function:** A loss function is a function that signifies how much our predicted values is deviated from the actual values of the dependent variable.

Important Note: *we are trying to find the values for **w** and **b** such that it minimizes our loss function.*

## Steps Involved in Linear Regression with Gradient Descent Implementation

1. Initialize the weight and bias randomly or with 0(both will work).
2. Make predictions with this initial weight and bias.
3. Compare these predicted values with the actual values and define the loss function using both these predicted and actual values.
4. With the help of differentiation, calculate how loss function changes with respect to weight and bias term.
5. Update the weight and bias term so as to minimize the loss function.

# Implementation with Math

| | | |
|---|---|---|
| 1 | 1.1 | 39343.00 |
| 2 | 1.3 | 46205.00 |
| 3 | 1.5 | 37731.00 |
| 4 | 2.0 | 43525.00 |
| 5 | 2.2 | 39891.00 |
| 6 | 2.9 | 56642.00 |
| 7 | 3.0 | 60150.00 |
| 8 | 3.2 | 54445.00 |
| 9 | 3.2 | 64445.00 |
| 10 | 3.7 | 57189.00 |

## 1 . Assumption

Let us say we have an **x** and **y** vectors like shown in the above pic (*The above one is only for an example*).

## 2. Initialize w and b to 0

**w** = 0, **b** = 0

## 3. Make some predictions with the current w and b. Of course, it's going to be wrong.

`y_pred = x*w + b`, where y_pred stands for predicted y values.
This y_pred will also be a vector like y.

## 4 . Define a loss function

`loss = (y_pred – y)²/n`
where n is the number of examples in the dataset.
It is obvious that this loss function represents the deviation of the predicted values from the actual.
This loss function will also be a vector. But, we will be summing all the elements in that vector to convert it into scalar.

## 5. Calculate (∂(loss)/ ∂w)

> *The derivative of a function of a real variable measures the sensitivity to change of the function value with respect to a change in its argument.*

We can use calculus to find how loss changes with respect to ***w***.
`loss = (y_pred – y)²/nloss = (y_pred² + y² – 2y*y_pred)/n` *(expanding the whole square)*`=>( (x*w+b)² + y² – 2`
`Let B = (y²/n),`
`Let C = ((-2y(x*w+b))/n)A = ( x²w² + b² + 2xwb )/n` *(expanding)*

```
∂A/∂w = ( 2x²w + 2xb )/n (differentiating)∂B/∂w = 0 (differentiating)C = (-2yxw – 2yb)/n
∂C/∂w = (-2yx)/n (differentiating)
```

So, ∂loss/∂w will be the addition of all these terms:

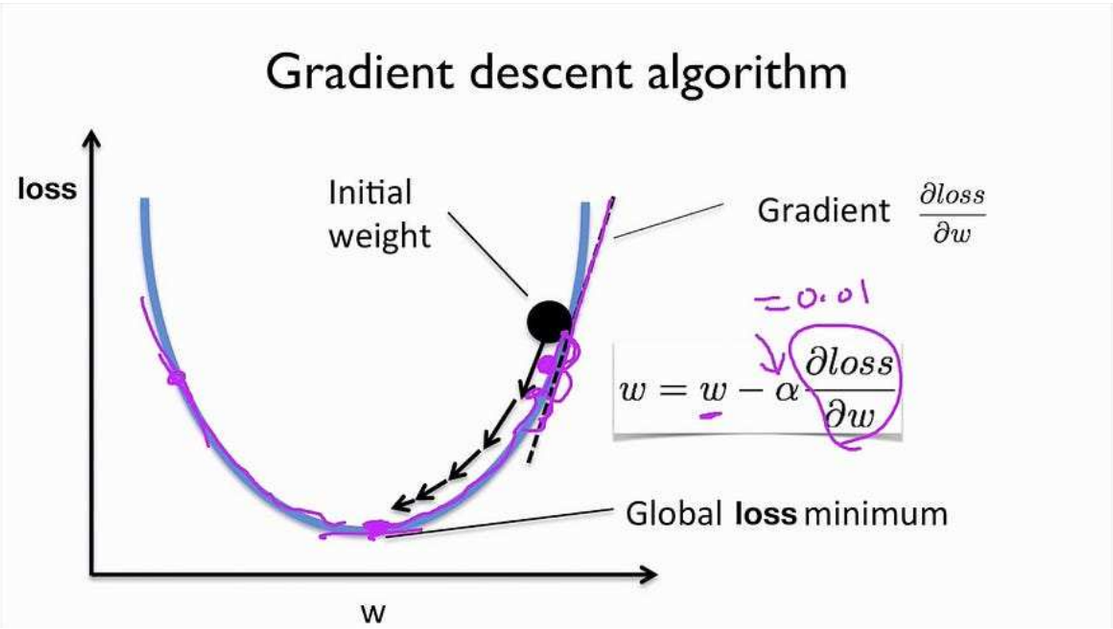
```
∂loss/∂w = (2x²w + wxb – 2yx)/n
=> (2x(x*w + b – y))/n
```

So, The derivative of loss with respect to ***w*** was found to be:
`(2/n)*(y_pred – y)*x`**.** Let us call this dw.

If we perform the same differentiation for loss with respect to ***b***, we'll get:
`(2/n)*(y_pred – y)`**.** Let us call this db.

This dw and db are what we call *"gradients"*

## 6. Update w and b




As we can see in the figure 6.1, if we initialize the weight randomly, it might not result in a global minimum of the loss function.
It is our duty to update the weights to the point where the loss is minimum. We have calculated **dw** above.

**dw** is nothing but the slope of the tangent of the loss function at point w.
Considering the initial position of **w**.

## Important point to understand :

**In the above diagram, The slope of the tangent of the loss will be positive as initial value of w is greater and it needs to be reduced so as to attain global minimum.**
**If the value of w is low and we want to increase it to attain global minimum, the slope of the tangent of loss at point w will be negative**

We want the value of w to be a little lower so as to attain global minima of the loss function as shown in the figure 6.1.

We know that dw is positive in the above graph and we need to reduce w.
This can be done by:

```
w = w – alpha*dw
b = b – alpha*bw
```

where alpha is a small number ranging between 0.1 to 0.0000001 (approx) and this alpha is knows as the learning rate.
*Doing this, we can reduce **w** if slope of tangent of loss at **w** is positive and increase **w** if slope is negative*

## 7. Learning Rate

Learning rate alpha is something that we have to manually choose and it is something which we don't know beforehand. Choosing it is a matter of trial and error.
The reason we do not directly subtract **dw** from **w** is because, it might result in too much change in the value of w and might not end up in global minimum but, even further away from it.

## 8. Training Loop

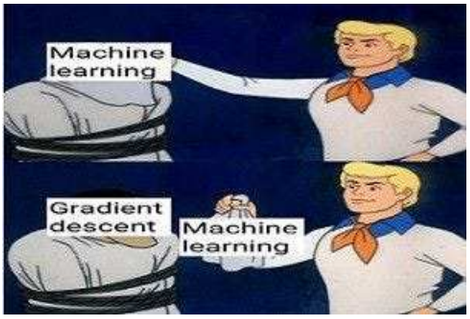
The process of calculating the gradients and updating the weight and bias is repeated for several times, which results in the optimized value for weight and bias.

## 9. Prediction

After the training loop, the value of weight and bias is now optimized, which can be used to predict new values given new x values.

```
y = x*w + b
```

# Conclusion

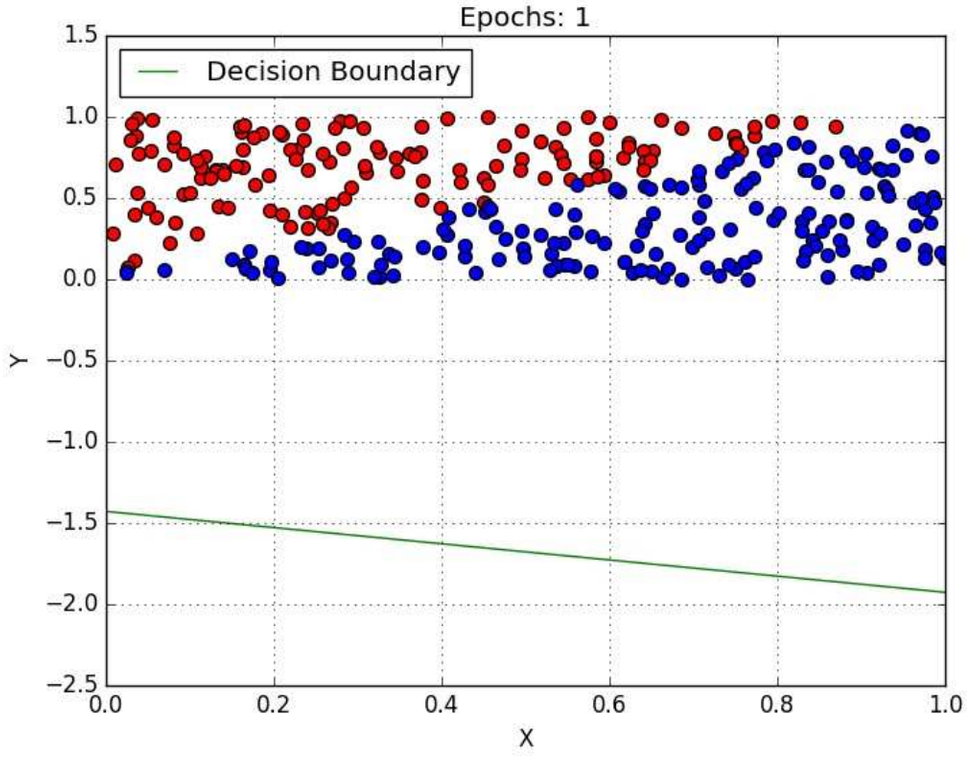



That's it for Linear Regression with gradient descent.
The learning from "Machine Learning" signifies the part where the gradients of **w** and **b** are learnt and then w and b are updated.

# Thank You

# 7. Logistic Regression

In this blog, we will discuss the basic concepts of Logistic Regression and what kind of problems can it help us to solve.



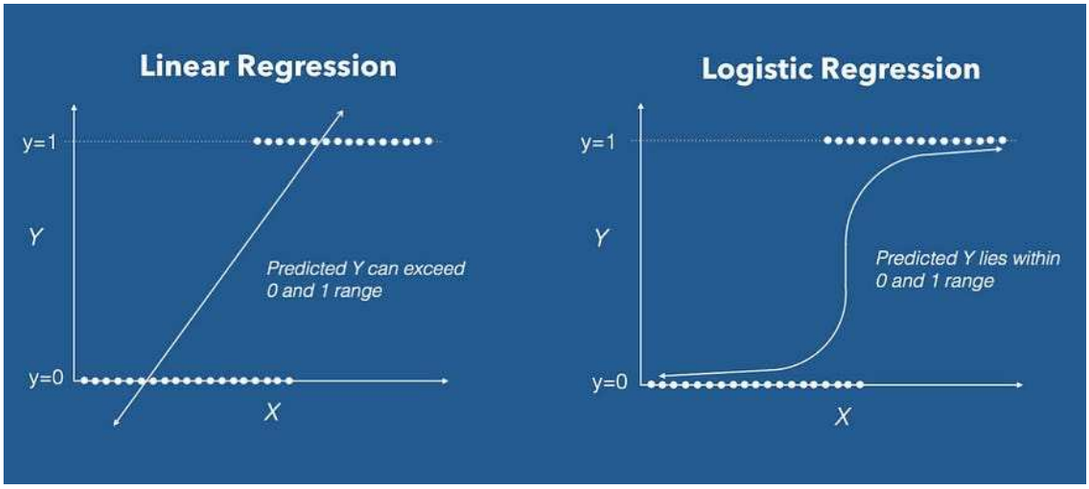

Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. Some of the examples of classification problems are Email spam or not spam, Online transactions Fraud or not Fraud, Tumor Malignant or Benign. Logistic regression transforms its output using the logistic sigmoid function to return a probability value.

## What are the types of logistic regression

1. Binary (eg. Tumor Malignant or Benign)
2. Multi-linear functions failsClass (eg. Cats, dogs or Sheep's)

# Logistic Regression

Logistic Regression is a Machine Learning algorithm which is used for the classification problems, it is a predictive analysis algorithm and based on the concept of probability.



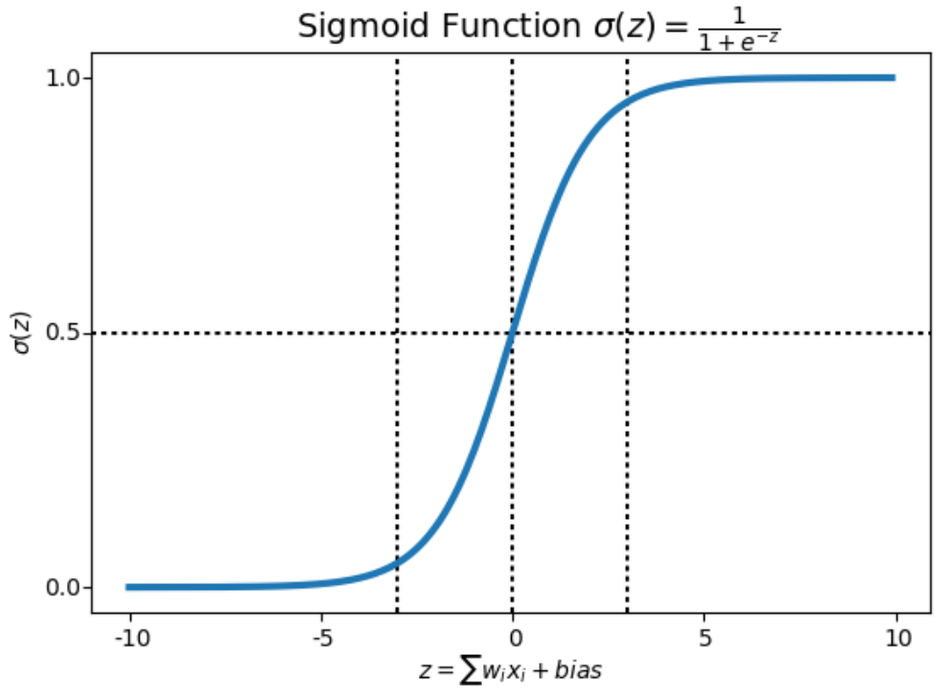

We can call a Logistic Regression a Linear Regression model but the Logistic Regression uses a more complex cost function, this cost function can be defined as the '**Sigmoid function**' or also known as the 'logistic function' instead of a linear function.

The hypothesis of logistic regression tends it to limit the cost function between 0 and 1. Therefore linear functions fail to represent it as it can have a value greater than 1 or less than 0 which is not possible as per the hypothesis of logistic regression.

$$0 \leq h_\theta(x) \leq 1$$

## What is the Sigmoid Function?

In order to map predicted values to probabilities, we use the Sigmoid function. The function maps any real value into another value between 0 and 1. In machine learning, we use sigmoid to map predictions to probabilities.




$$f(x) = \frac{1}{1+e^{-(x)}}$$

# Hypothesis Representation

When using *linear regression* we used a formula of the hypothesis i.e.

$h\Theta(x) = \beta_0 + \beta_1 X$

For logistic regression we are going to modify it a little bit i.e.

$\sigma(Z) = \sigma(\beta_0 + \beta_1 X)$

We have expected that our hypothesis will give values between 0 and 1.

$Z = \beta_0 + \beta_1 X$

$h\Theta(x) = sigmoid(Z)$

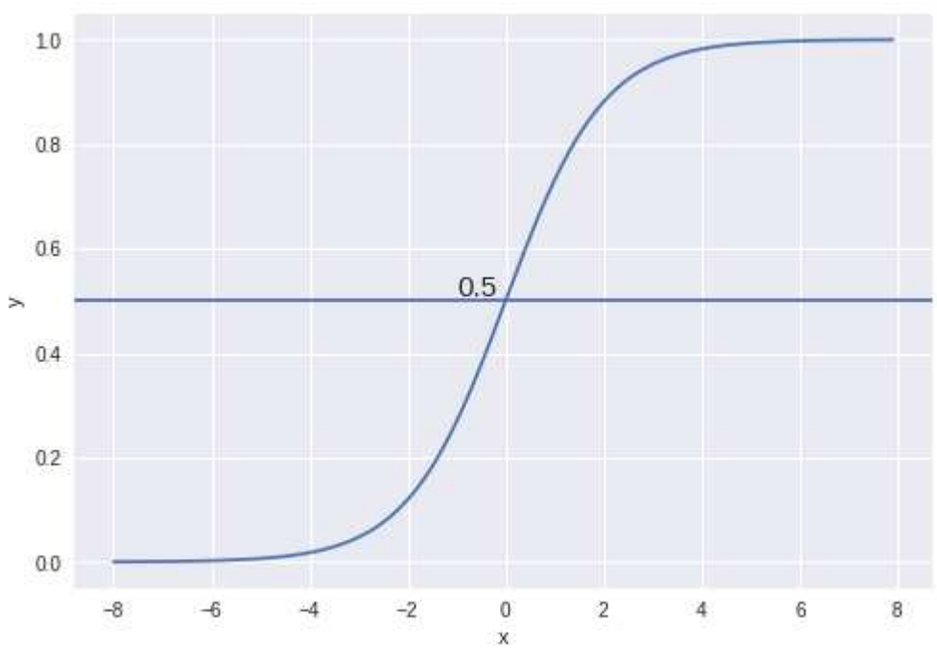
i.e. $h\Theta(x) = 1/(1 + e^{-(\beta_0 + \beta_1 X)}$

$$h\theta(X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

## Decision Boundary

We expect our classifier to give us a set of outputs or classes based on probability when we pass the inputs through a prediction function and returns a probability score between 0 and 1.

For Example, We have 2 classes, let's take them like cats and dogs(1 — dog , 0 — cats). We basically decide with a threshold value above which we classify values into Class 1 and of the value goes below the threshold then we classify it in Class 2.



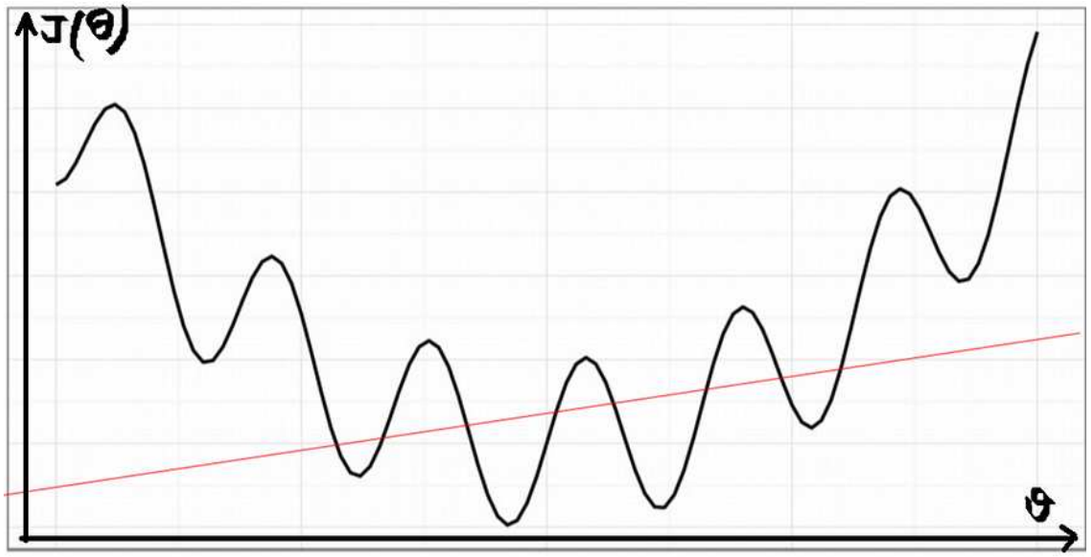

As shown in the above graph we have chosen the threshold as 0.5, if the prediction function returned a value of 0.7 then we would classify this observation as Class 1(DOG). If our prediction returned a value of 0.2 then we would classify the observation as Class 2(CAT).

# Cost Function

We learnt about the cost function $J(\theta)$ in the *Linear regression*, the cost function represents optimization objective i.e. we create a cost function and minimize it so that we can develop an accurate model with minimum error.

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2.$$

If we try to use the cost function of the linear regression in 'Logistic Regression' then it would be of no use as it would end up being a **non-convex** function with many local minimums, in which it would be very **difficult** to **minimize the cost value** and find the global minimum.




For logistic regression, the Cost function is defined as:

*−log(hθ(x)) if y = 1*

*−log(1−hθ(x)) if y = 0*

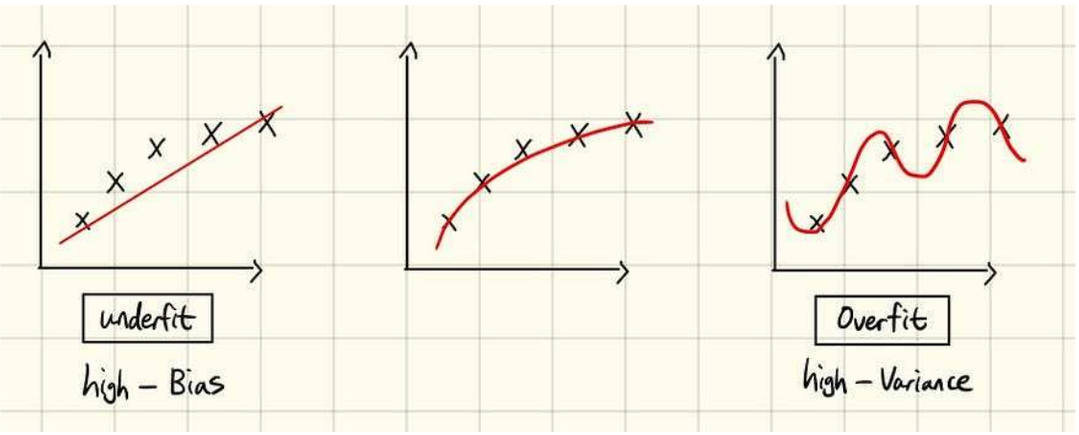
$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y = 1 \\ -log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

# 7.2 Overfitting

Overfitting & How to solve it-Andrew Ng

## The Problem of Overfitting

Having too many features can cause a problem. The hypothesis function can become too complicated. It would predict the training example very well, but our purpose is to perform prediction on a new data, a data not included in the training set. Hypothesis that is too fit to the training data can be very poor when predicting a new data. When a hypothesis is too fit to the training data, we address this problem as “Overfitting”.




- Bias: The accuracy of our predictions
- Variance: The difference between many model’s predictions

If we have high bias, we cannot predict the data well so we have an “underfit” situation. In contrast, if we have high variance, it can be told that the model is not generalized well and would perform poorly on a new data. We have an “overfit” situation.

## Options to deal with overfitting

There are some options to deal with overfitting.

- Option 1: Reduce feature number

Manually select features you want to use/Model selection algorithm

However, you might not want to throw away the features. Maybe all of the features might be a help to our prediction. In this case, we retain all of the features and try to avoid overfitting. This method is called "Regularization".

- Option 2: Regularization

Retain all of the features/reduce the magnitude(value) of parameter theta

■

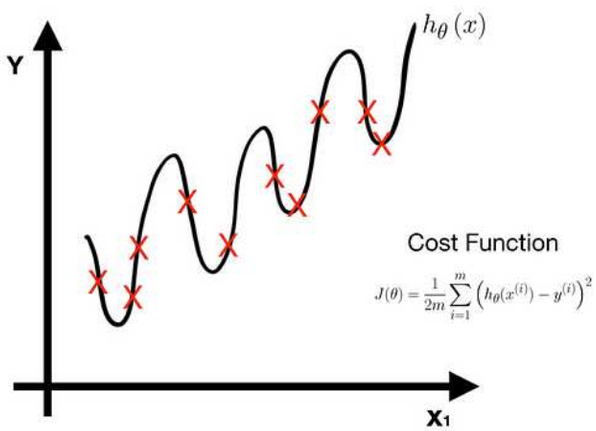
# 7.3 Overfitting and Regularization

After learning process, we get a good model. If we apply it to certain problems, we may or may not get good performance. That is the issue we are going to discuss today — How come the performance of a good model is terrible?

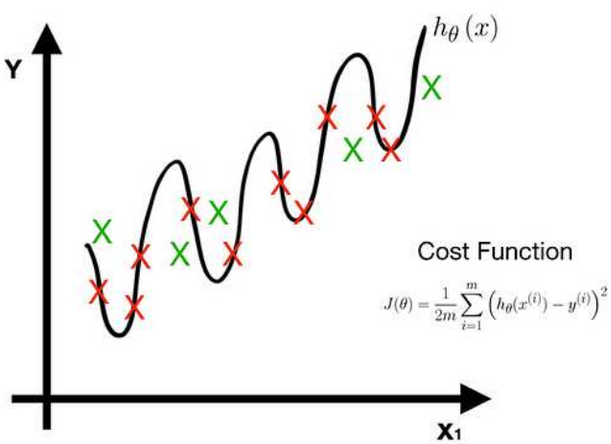
Let's think it first— what is a good model?

In the previous note, the definition of a 'good' model is the one with least training error. First, define a hypothesis function ***H*** (i.e. model). Second, define a Cost Function ***J(θ)*** for error measuring. Third, learn parameters of ***H*** by minimizing ***J(θ)***. In the end, we can get a learned model with least training error. In the extreme case, this 'good' model has zero training error and fits training data perfectly, that is, the predict label of all training data is equal to their corresponding truth label.

Taking linear regression as example, the model fits all training data (red points) perfectly.




However, when new data (green point) comes in, this model has terrible performance with huge prediction error.



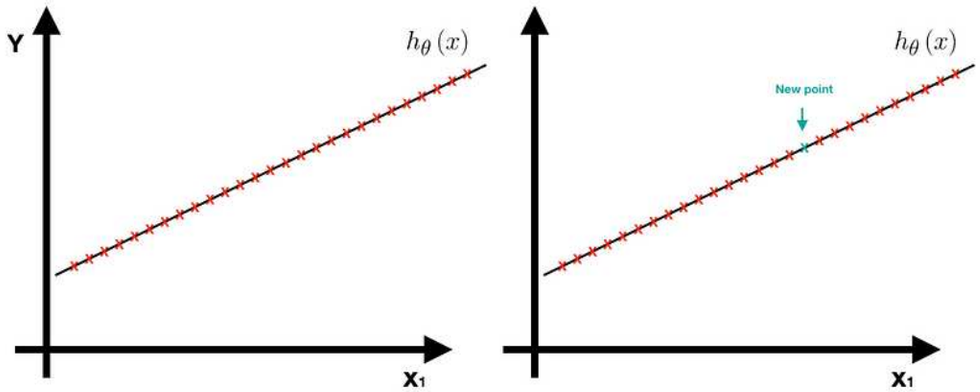

The problem here is called Overfitting. A truly good model must have both little training error and little prediction error.

# Overfitting

> *The learned model works well for training data but terrible for testing data (unknown data). In other words, the model has little training error but has huge perdition error.*

If we have a whole dataset that captured all possibilities of the problem, we don't need to worry about overfitting. When new data comes in, it shall fall into one possibility, hence, the model can predict it perfectly.

For example, suppose we have a training dataset that captures all points on the line and also get a perfect model h(x) with zero training error. Whenever new data comes in, it shall be one of the points, hence, the prediction error must be zero. In this special case, overfitting is not a problem.




Unfortunately, the training data we get in reality is normally a small part of the whole dataset. Therefore, even if the model fits these training data perfectly, this is still not a good model and overfitting must occur.

When overfitting occurs, we get an over complex model with too many features. One way to avoid it is to apply Regularization and then we can get a better model with proper features.

# Regularization

> *It's a technique applied to Cost Function **J(θ)** in order to avoid Overfitting.*

The core idea in Regularization is to keep more important features and ignore unimportant ones. The importance of feature is measured by the value of its parameter ***θj.***

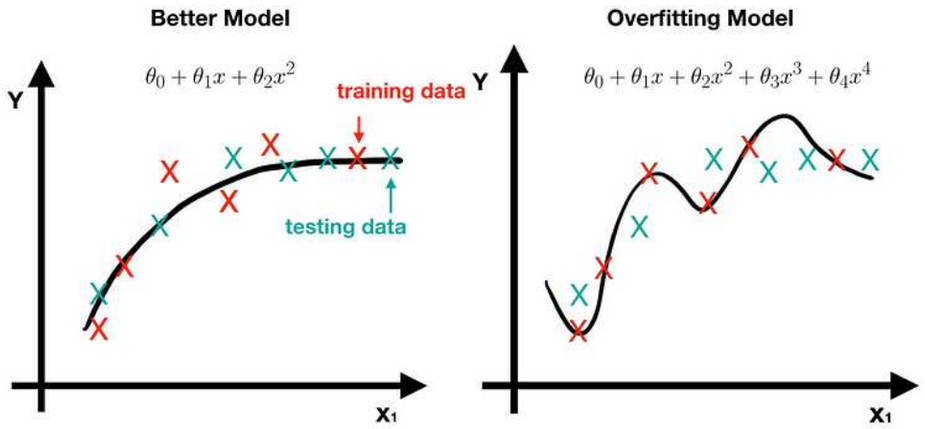
In linear regression, we modify its cost function by adding regularization term. The value of ***θj*** is controlled by regularization parameter **λ**. Note that ***m*** is the number of data and ***n*** is the number of features(parameters.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \underbrace{\boxed{\lambda}\sum_{j=1}^{n}\theta_j^{\,2}}_{\text{Regularization Term}}$$

start at $\theta_1$

Regularization Parameter

For instance, if we want to get a better model instead of the overfitting one. Obviously, we don't need features $X^3$ and $X^4$ since they are unimportant. The procedure describes below.




First, we modify the Cost Function ***J(θ)*** by adding regularization. Second, apply gradient descent in order to minimize ***J(θ)*** and get the values of ***θ3*** and ***θ4.*** After the minimize procedure, the values of ***θ3*** and ***θ4*** must be near to zero if **λ=1000.**

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \underbrace{\boxed{1000{\theta_3}^2 + 1000{\theta_4}^2}}_{\text{Regularization Term}}$$

$$\underset{\theta}{Min}\ J(\theta),\ getting\ \theta_3 \approx 0,\ \theta_4 \approx 0$$

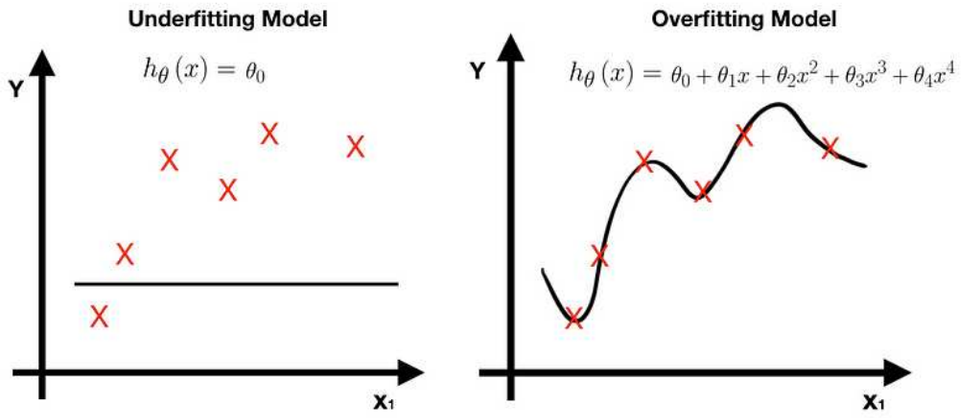
Remember, the value of ***J(θ)*** represents training error and this value must be positive (≥0). The parameter **λ=1000** has significant effect on ***J(θ),*** therefore, ***θ3*** and ***θ4*** must be near to zero (e.g 0.000001) so as to eliminate error value.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\boxed{\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2} + 1000\boxed{{\theta_3}^2} + 1000\boxed{{\theta_4}^2}$$

Positive

## Regularization Parameter λ

- If **λ** is too large, then all the values of ***θ*** may be near to zero and this may cause Underfitting. In other words, this model has both large training error and large prediction error. (Note that the regularization term starts from ***θ1***)
- If **λ** is zero or too small, its effect on parameters ***θ*** is little. This may cause Overfitting.




To sum up, there are two advantages of using regularization.

- The prediction error of the regularized model is lesser, that is, it works well in testing data (green points).
- The regularization model is simpler since it has less features (parameters).

# CHAPTER 3

# DECISION TREE LEARNING

Decision tree learning is one of the most widely used and practical methods for inductive inference. It is a method for approximating discrete-valued functions that is robust to noisy data and capable of learning disjunctive expressions. This chapter describes a family of decision tree learning algorithms that includes widely used algorithms such as ID3, ASSISTANT, and C4.5. These decision tree learning methods search a completely expressive hypothesis space and thus avoid the difficulties of restricted hypothesis spaces. Their inductive bias is a preference for small trees over large trees.

## 3.1 INTRODUCTION

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. Learned trees can also be re-represented as sets of if-then rules to improve human readability. These learning methods are among the most popular of inductive inference algorithms and have been successfully applied to a broad range of tasks from learning to diagnose medical cases to learning to assess credit risk of loan applicants.

## 3.2 DECISION TREE REPRESENTATION

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some *attribute* of the instance, and each branch descending

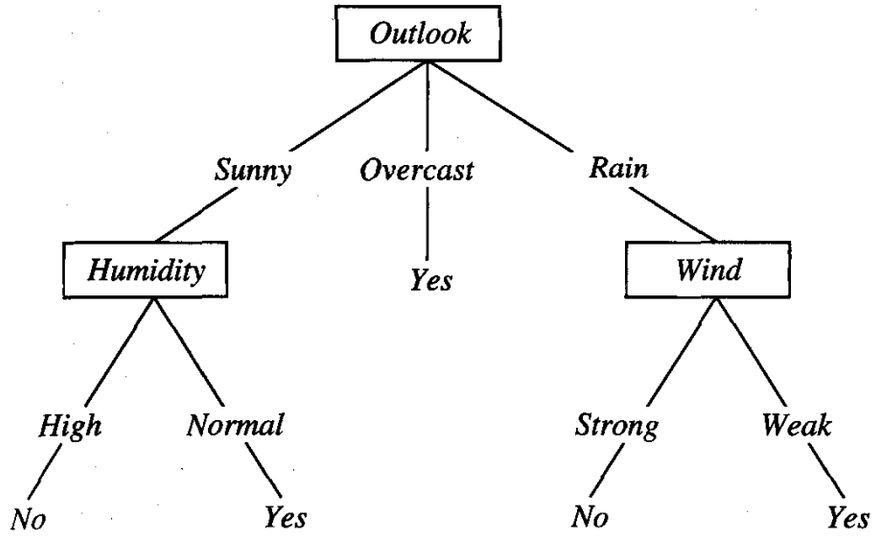



**FIGURE 3.1**
A decision tree for the concept *PlayTennis*. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf (in this case, *Yes* or *No*). This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis.

from that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

Figure 3.1 illustrates a typical learned decision tree. This decision tree classifies Saturday mornings according to whether they are suitable for playing tennis. For example, the instance

$$\langle Outlook = Sunny,\ Temperature = Hot,\ Humidity = High,\ Wind = Strong \rangle$$

would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance (i.e., the tree predicts that $PlayTennis = no$). This tree and the example used in Table 3.2 to illustrate the ID3 learning algorithm are adapted from (Quinlan 1986).

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions. For example, the decision tree shown in Figure 3.1 corresponds to the expression

$$\begin{aligned} & (Outlook = Sunny \wedge Humidity = Normal) \\ \vee\ & (Outlook = Overcast) \\ \vee\ & (Outlook = Rain \wedge Wind = Weak) \end{aligned}$$

## 3.3 APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

Although a variety of decision tree learning methods have been developed with somewhat differing capabilities and requirements, decision tree learning is generally best suited to problems with the following characteristics:

- *Instances are represented by attribute-value pairs.* Instances are described by a fixed set of attributes (e.g., *Temperature*) and their values (e.g., *Hot*). The easiest situation for decision tree learning is when each attribute takes on a small number of disjoint possible values (e.g., *Hot*, *Mild*, *Cold*). However, extensions to the basic algorithm (discussed in Section 3.7.2) allow handling real-valued attributes as well (e.g., representing *Temperature* numerically).
- *The target function has discrete output values.* The decision tree in Figure 3.1 assigns a boolean classification (e.g., *yes* or *no*) to each example. Decision tree methods easily extend to learning functions with more than two possible output values. A more substantial extension allows learning target functions with real-valued outputs, though the application of decision trees in this setting is less common.
- *Disjunctive descriptions may be required.* As noted above, decision trees naturally represent disjunctive expressions.
- *The training data may contain errors.* Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.
- *The training data may contain missing attribute values.* Decision tree methods can be used even when some training examples have unknown values (e.g., if the *Humidity* of the day is known for only some of the training examples). This issue is discussed in Section 3.7.4.

Many practical problems have been found to fit these characteristics. Decision tree learning has therefore been applied to problems such as learning to classify medical patients by their disease, equipment malfunctions by their cause, and loan applicants by their likelihood of defaulting on payments. Such problems, in which the task is to classify examples into one of a discrete set of possible categories, are often referred to as *classification problems*.

The remainder of this chapter is organized as follows. Section 3.4 presents the basic ID3 algorithm for learning decision trees and illustrates its operation in detail. Section 3.5 examines the hypothesis space search performed by this learning algorithm, contrasting it with algorithms from Chapter 2. Section 3.6 characterizes the inductive bias of this decision tree learning algorithm and explores more generally an inductive bias called Occam's razor, which corresponds to a preference for the most simple hypothesis. Section 3.7 discusses the issue of overfitting the training data, as well as strategies such as rule post-pruning to deal with this problem. This section also discusses a number of more advanced topics such as extending the algorithm to accommodate real-valued attributes, training data with unobserved attributes, and attributes with differing costs.

## 3.4 THE BASIC DECISION TREE LEARNING ALGORITHM

Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees. This approach is exemplified by the ID3 algorithm (Quinlan 1986) and its successor C4.5 (Quinlan 1993), which form the primary focus of our discussion here. In this section we present the basic algorithm for decision tree learning, corresponding approximately to the ID3 algorithm. In Section 3.7 we consider a number of extensions to this basic algorithm, including extensions incorporated into C4.5 and other more recent algorithms for decision tree learning.

Our basic algorithm, ID3, learns decision trees by constructing them top-down, beginning with the question "which attribute should be tested at the root of the tree?" To answer this question, each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples. The best attribute is selected and used as the test at the root node of the tree. A descendant of the root node is then created for each possible value of this attribute, and the training examples are sorted to the appropriate descendant node (i.e., down the branch corresponding to the example's value for this attribute). The entire process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that point in the tree. This forms a greedy search for an acceptable decision tree, in which the algorithm never backtracks to reconsider earlier choices. A simplified version of the algorithm, specialized to learning boolean-valued functions (i.e., concept learning), is described in Table 3.1.

### 3.4.1 Which Attribute Is the Best Classifier?

The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree. We would like to select the attribute that is most useful for classifying examples. What is a good quantitative measure of the worth of an attribute? We will define a statistical property, called *information gain*, that measures how well a given attribute separates the training examples according to their target classification. ID3 uses this information gain measure to select among the candidate attributes at each step while growing the tree.

#### 3.4.1.1 ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

In order to define information gain precisely, we begin by defining a measure commonly used in information theory, called *entropy*, that characterizes the (im)purity of an arbitrary collection of examples. Given a collection $S$, containing positive and negative examples of some target concept, the entropy of $S$ relative to this boolean classification is

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus} \tag{3.1}$$

ID3(*Examples*, *Target_attribute*, *Attributes*)

*Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.*

- Create a *Root* node for the tree
- If all *Examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Examples* are negative, Return the single-node tree *Root*, with label = −
- If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Examples*
- Otherwise Begin
  - $A \leftarrow$ the attribute from *Attributes* that best* classifies *Examples*
  - The decision attribute for *Root* $\leftarrow A$
  - For each possible value, $v_i$, of $A$,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ be the subset of *Examples* that have value $v_i$ for $A$
    - If $Examples_{v_i}$ is empty
      - Then below this new branch add a leaf node with label = most common value of *Target_attribute* in *Examples*
      - Else below this new branch add the subtree ID3($Examples_{v_i}$, *Target_attribute*, *Attributes* − {$A$}))
- End
- Return *Root*

* The best attribute is the one with highest *information gain*, as defined in Equation (3.4).

**TABLE 3.1**
Summary of the ID3 algorithm specialized to learning boolean-valued functions. ID3 is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples, or until all attributes have been used.

where $p_\oplus$ is the proportion of positive examples in $S$ and $p_\ominus$ is the proportion of negative examples in $S$. In all calculations involving entropy we define $0 \log 0$ to be 0.

To illustrate, suppose $S$ is a collection of 14 examples of some boolean concept, including 9 positive and 5 negative examples (we adopt the notation [9+, 5−] to summarize such a sample of data). Then the entropy of $S$ relative to this boolean classification is

$$\begin{aligned} Entropy([9+, 5-]) &= -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) \\ &= 0.940 \end{aligned} \tag{3.2}$$

Notice that the entropy is 0 if all members of S belong to the same class. For example, if all members are positive ($p_\oplus = 1$), then $p_\ominus$ is 0, and $Entropy(S) = -1 \cdot \log_2(1) - 0 \cdot \log_2 0 = -1 \cdot 0 - 0 \cdot \log_2 0 = 0$. Note the entropy is 1 when the collection contains an equal number of positive and negative examples. If the collection contains unequal numbers of positive and negative examples, the

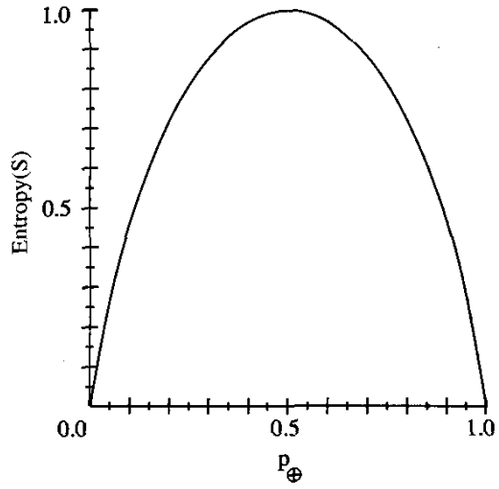



**FIGURE 3.2**
The entropy function relative to a boolean classification, as the proportion, $p_\oplus$, of positive examples varies between 0 and 1.

entropy is between 0 and 1. Figure 3.2 shows the form of the entropy function relative to a boolean classification, as $p_\oplus$ varies between 0 and 1.

One interpretation of entropy from information theory is that it specifies the minimum number of bits of information needed to encode the classification of an arbitrary member of $S$ (i.e., a member of $S$ drawn at random with uniform probability). For example, if $p_\oplus$ is 1, the receiver knows the drawn example will be positive, so no message need be sent, and the entropy is zero. On the other hand, if $p_\oplus$ is 0.5, one bit is required to indicate whether the drawn example is positive or negative. If $p_\oplus$ is 0.8, then a collection of messages can be encoded using on average less than 1 bit per message by assigning shorter codes to collections of positive examples and longer codes to less likely negative examples.

Thus far we have discussed entropy in the special case where the target classification is boolean. More generally, if the target attribute can take on $c$ different values, then the entropy of $S$ relative to this $c$-wise classification is defined as

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i \tag{3.3}$$

where $p_i$ is the proportion of $S$ belonging to class $i$. Note the logarithm is still base 2 because entropy is a measure of the expected encoding length measured in *bits*. Note also that if the target attribute can take on $c$ possible values, the entropy can be as large as $\log_2 c$.

### 3.4.1.2 INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

Given entropy as a measure of the impurity in a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called *information gain*, is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, $Gain(S, A)$ of an attribute $A$,

relative to a collection of examples $S$, is defined as

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{3.4}$$

where $Values(A)$ is the set of all possible values for attribute $A$, and $S_v$ is the subset of $S$ for which attribute $A$ has value $v$ (i.e., $S_v = \{s \in S | A(s) = v\}$). Note the first term in Equation (3.4) is just the entropy of the original collection $S$, and the second term is the expected value of the entropy after $S$ is partitioned using attribute $A$. The expected entropy described by this second term is simply the sum of the entropies of each subset $S_v$, weighted by the fraction of examples $\frac{|S_v|}{|S|}$ that belong to $S_v$. $Gain(S, A)$ is therefore the expected reduction in entropy caused by knowing the value of attribute $A$. Put another way, $Gain(S, A)$ is the information provided about the *target function value*, given the value of some other attribute $A$. The value of $Gain(S, A)$ is the number of bits saved when encoding the target value of an arbitrary member of $S$, by knowing the value of attribute $A$.

For example, suppose $S$ is a collection of training-example days described by attributes including *Wind*, which can have the values *Weak* or *Strong*. As before, assume $S$ is a collection containing 14 examples, [9+, 5−]. Of these 14 examples, suppose 6 of the positive and 2 of the negative examples have *Wind* = *Weak*, and the remainder have *Wind* = *Strong*. The information gain due to sorting the original 14 examples by the attribute *Wind* may then be calculated as

$$\begin{aligned}
Values(Wind) &= Weak, Strong \\
S &= [9+, 5-] \\
S_{Weak} &\leftarrow [6+, 2-] \\
S_{Strong} &\leftarrow [3+, 3-] \\
Gain(S, Wind) &= Entropy(S) - \sum_{v \in \{Weak, Strong\}} \frac{|S_v|}{|S|} Entropy(S_v) \\
&= Entropy(S) - (8/14) Entropy(S_{Weak}) \\
&\quad - (6/14) Entropy(S_{Strong}) \\
&= 0.940 - (8/14)0.811 - (6/14)1.00 \\
&= 0.048
\end{aligned}$$

Information gain is precisely the measure used by ID3 to select the best attribute at each step in growing the tree. The use of information gain to evaluate the relevance of attributes is summarized in Figure 3.3. In this figure the information gain of two different attributes, *Humidity* and *Wind*, is computed in order to determine which is the better attribute for classifying the training examples shown in Table 3.2.

**Which attribute is the best classifier?**

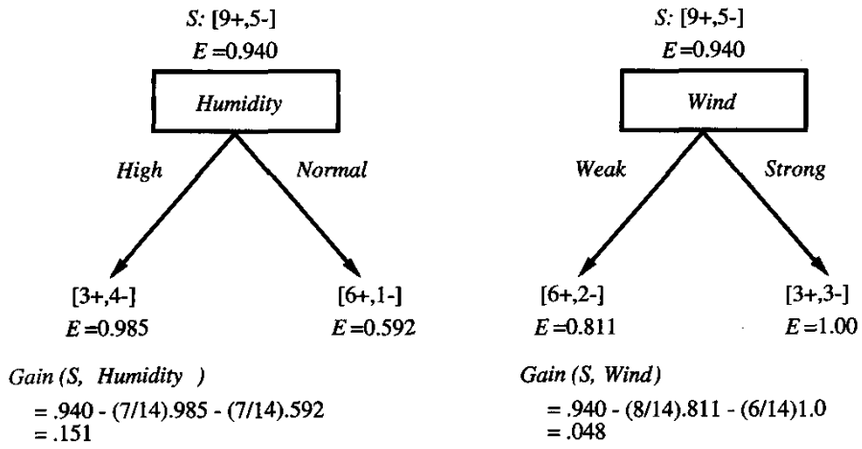



**FIGURE 3.3**
*Humidity* provides greater information gain than *Wind*, relative to the target classification. Here, $E$ stands for entropy and $S$ for the original collection of examples. Given an initial collection $S$ of 9 positive and 5 negative examples, [9+, 5−], sorting these by their *Humidity* produces collections of [3+, 4−] ($Humidity = High$) and [6+, 1−] ($Humidity = Normal$). The information gained by this partitioning is .151, compared to a gain of only .048 for the attribute *Wind*.

## 3.4.2 An Illustrative Example

To illustrate the operation of ID3, consider the learning task represented by the training examples of Table 3.2. Here the target attribute *PlayTennis*, which can have values *yes* or *no* for different Saturday mornings, is to be predicted based on other attributes of the morning in question. Consider the first step through

| Day | *Outlook* | *Temperature* | *Humidity* | *Wind* | *PlayTennis* |
|---|---|---|---|---|---|
| D1 | Sunny | Hot | High | Weak | No |
| D2 | Sunny | Hot | High | Strong | No |
| D3 | Overcast | Hot | High | Weak | Yes |
| D4 | Rain | Mild | High | Weak | Yes |
| D5 | Rain | Cool | Normal | Weak | Yes |
| D6 | Rain | Cool | Normal | Strong | No |
| D7 | Overcast | Cool | Normal | Strong | Yes |
| D8 | Sunny | Mild | High | Weak | No |
| D9 | Sunny | Cool | Normal | Weak | Yes |
| D10 | Rain | Mild | Normal | Weak | Yes |
| D11 | Sunny | Mild | Normal | Strong | Yes |
| D12 | Overcast | Mild | High | Strong | Yes |
| D13 | Overcast | Hot | Normal | Weak | Yes |
| D14 | Rain | Mild | High | Strong | No |

**TABLE 3.2**
Training examples for the target concept *PlayTennis*.

the algorithm, in which the topmost node of the decision tree is created. Which attribute should be tested first in the tree? ID3 determines the information gain for each candidate attribute (i.e., *Outlook, Temperature, Humidity*, and *Wind*), then selects the one with highest information gain. The computation of information gain for two of these attributes is shown in Figure 3.3. The information gain values for all four attributes are

$$Gain(S, Outlook) = 0.246$$

$$Gain(S, Humidity) = 0.151$$

$$Gain(S, Wind) = 0.048$$

$$Gain(S, Temperature) = 0.029$$

where $S$ denotes the collection of training examples from Table 3.2.

According to the information gain measure, the *Outlook* attribute provides the best prediction of the target attribute, *PlayTennis*, over the training examples. Therefore, *Outlook* is selected as the decision attribute for the root node, and branches are created below the root for each of its possible values (i.e., *Sunny*, *Overcast*, and *Rain*). The resulting partial decision tree is shown in Figure 3.4, along with the training examples sorted to each new descendant node. Note that every example for which *Outlook* = *Overcast* is also a positive example of *PlayTennis*. Therefore, this node of the tree becomes a leaf node with the classification *PlayTennis* = *Yes*. In contrast, the descendants corresponding to *Outlook* = *Sunny* and *Outlook* = *Rain* still have nonzero entropy, and the decision tree will be further elaborated below these nodes.

The process of selecting a new attribute and partitioning the training examples is now repeated for each nonterminal descendant node, this time using only the training examples associated with that node. Attributes that have been incorporated higher in the tree are excluded, so that any given attribute can appear at most once along any path through the tree. This process continues for each new leaf node until either of two conditions is met: (1) every attribute has already been included along this path through the tree, or (2) the training examples associated with this leaf node all have the same target attribute value (i.e., their entropy is zero). Figure 3.4 illustrates the computations of information gain for the next step in growing the decision tree. The final decision tree learned by ID3 from the 14 training examples of Table 3.2 is shown in Figure 3.1.

## 3.5 HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

As with other inductive learning methods, ID3 can be characterized as searching a space of hypotheses for one that fits the training examples. The hypothesis space searched by ID3 is the set of possible decision trees. ID3 performs a simple-to-complex, hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data. The evaluation function

## 3.6 INDUCTIVE BIAS IN DECISION TREE LEARNING

What is the policy by which ID3 generalizes from observed training examples to classify unseen instances? In other words, what is its inductive bias? Recall from Chapter 2 that inductive bias is the set of assumptions that, together with the training data, deductively justify the classifications assigned by the learner to future instances.

Given a collection of training examples, there are typically many decision trees consistent with these examples. Describing the inductive bias of ID3 therefore consists of describing the basis by which it chooses one of these consistent hypotheses over the others. Which of these decision trees does ID3 choose? It chooses the first acceptable tree it encounters in its simple-to-complex, hill-climbing search through the space of possible trees. Roughly speaking, then, the ID3 search strategy (a) selects in favor of shorter trees over longer ones, and (b) selects trees that place the attributes with highest information gain closest to the root. Because of the subtle interaction between the attribute selection heuristic used by ID3 and the particular training examples it encounters, it is difficult to characterize precisely the inductive bias exhibited by ID3. However, we can approximately characterize its bias as a preference for short decision trees over complex trees.

**Approximate inductive bias of ID3:** Shorter trees are preferred over larger trees.

In fact, one could imagine an algorithm similar to ID3 that exhibits precisely this inductive bias. Consider an algorithm that begins with the empty tree and searches *breadth first* through progressively more complex trees, first considering all trees of depth 1, then all trees of depth 2, etc. Once it finds a decision tree consistent with the training data, it returns the smallest consistent tree at that search depth (e.g., the tree with the fewest nodes). Let us call this breadth-first search algorithm BFS-ID3. BFS-ID3 finds a shortest decision tree and thus exhibits precisely the bias "shorter trees are preferred over longer trees." ID3 can be viewed as an efficient approximation to BFS-ID3, using a greedy heuristic search to attempt to find the shortest tree without conducting the entire breadth-first search through the hypothesis space.

Because ID3 uses the information gain heuristic and a hill climbing strategy, it exhibits a more complex bias than BFS-ID3. In particular, it does not always find the shortest consistent tree, and it is biased to favor trees that place attributes with high information gain closest to the root.

**A closer approximation to the inductive bias of ID3:** Shorter trees are preferred over longer trees. Trees that place high information gain attributes close to the root are preferred over those that do not.

### 3.6.1 Restriction Biases and Preference Biases

There is an interesting difference between the types of inductive bias exhibited by ID3 and by the CANDIDATE-ELIMINATION algorithm discussed in Chapter 2.

Consider the difference between the hypothesis space search in these two approaches:

- ID3 searches a *complete* hypothesis space (i.e., one capable of expressing any finite discrete-valued function). It searches *incompletely* through this space, from simple to complex hypotheses, until its termination condition is met (e.g., until it finds a hypothesis consistent with the data). Its inductive bias is solely a consequence of the ordering of hypotheses by its search strategy. Its hypothesis space introduces no additional bias.
- The version space CANDIDATE-ELIMINATION algorithm searches an *incomplete* hypothesis space (i.e., one that can express only a subset of the potentially teachable concepts). It searches this space *completely*, finding every hypothesis consistent with the training data. Its inductive bias is solely a consequence of the expressive power of its hypothesis representation. Its search strategy introduces no additional bias.

In brief, the inductive bias of ID3 follows from its *search strategy*, whereas the inductive bias of the CANDIDATE-ELIMINATION algorithm follows from the definition of its *search space*.

The inductive bias of ID3 is thus a *preference* for certain hypotheses over others (e.g., for shorter hypotheses), with no hard restriction on the hypotheses that can be eventually enumerated. This form of bias is typically called a *preference bias* (or, alternatively, a *search bias*). In contrast, the bias of the CANDIDATE-ELIMINATION algorithm is in the form of a categorical *restriction* on the set of hypotheses considered. This form of bias is typically called a *restriction bias* (or, alternatively, a *language bias*).

Given that some form of inductive bias is required in order to generalize beyond the training data (see Chapter 2), which type of inductive bias shall we prefer; a preference bias or restriction bias?

Typically, a preference bias is more desirable than a restriction bias, because it allows the learner to work within a complete hypothesis space that is assured to contain the unknown target function. In contrast, a restriction bias that strictly limits the set of potential hypotheses is generally less desirable, because it introduces the possibility of excluding the unknown target function altogether.

Whereas ID3 exhibits a purely preference bias and CANDIDATE-ELIMINATION a purely restriction bias, some learning systems combine both. Consider, for example, the program described in Chapter 1 for learning a numerical evaluation function for game playing. In this case, the learned evaluation function is represented by a linear combination of a fixed set of board features, and the learning algorithm adjusts the parameters of this linear combination to best fit the available training data. In this case, the decision to use a linear function to represent the evaluation function introduces a restriction bias (nonlinear evaluation functions cannot be represented in this form). At the same time, the choice of a particular parameter tuning method (the LMS algorithm in this case) introduces a preference bias stemming from the ordered search through the space of all possible parameter values.

### 3.6.2 Why Prefer Short Hypotheses?

Is ID3's inductive bias favoring shorter decision trees a sound basis for generalizing beyond the training data? Philosophers and others have debated this question for centuries, and the debate remains unresolved to this day. William of Occam was one of the first to discuss† the question, around the year 1320, so this bias often goes by the name of Occam's razor.

**Occam's razor:** Prefer the simplest hypothesis that fits the data.

Of course giving an inductive bias a name does not justify it. Why should one prefer simpler hypotheses? Notice that scientists sometimes appear to follow this inductive bias. Physicists, for example, prefer simple explanations for the motions of the planets, over more complex explanations. Why? One argument is that because there are fewer short hypotheses than long ones (based on straightforward combinatorial arguments), it is less likely that one will find a short hypothesis that coincidentally fits the training data. In contrast there are often many very complex hypotheses that fit the current training data but fail to generalize correctly to subsequent data. Consider decision tree hypotheses, for example. There are many more 500-node decision trees than 5-node decision trees. Given a small set of 20 training examples, we might expect to be able to find many 500-node decision trees consistent with these, whereas we would be more surprised if a 5-node decision tree could perfectly fit this data. We might therefore believe the 5-node tree is less likely to be a statistical coincidence and prefer this hypothesis over the 500-node hypothesis.

Upon closer examination, it turns out there is a major difficulty with the above argument. By the same reasoning we could have argued that one should prefer decision trees containing exactly 17 leaf nodes with 11 nonleaf nodes, that use the decision attribute $A_1$ at the root, and test attributes $A_2$ through $A_{11}$, in numerical order. There are relatively few such trees, and we might argue (by the same reasoning as above) that our a priori chance of finding one consistent with an arbitrary set of data is therefore small. The difficulty here is that there are very many small sets of hypotheses that one can define—most of them rather arcane. Why should we believe that the small set of hypotheses consisting of decision trees with *short descriptions* should be any more relevant than the multitude of other small sets of hypotheses that we might define?

A second problem with the above argument for Occam's razor is that the size of a hypothesis is determined by the particular representation used *internally* by the learner. Two learners using different internal representations could therefore arrive at different hypotheses, both justifying their contradictory conclusions by Occam's razor! For example, the function represented by the learned decision tree in Figure 3.1 could be represented as a tree with just one decision node, by a learner that uses the boolean attribute $XYZ$, where we define the attribute $XYZ$ to

† Apparently while shaving.

be true for instances that are classified positive by the decision tree in Figure 3.1 and false otherwise. Thus, two learners, both applying Occam's razor, would generalize in different ways if one used the $XYZ$ attribute to describe its examples and the other used only the attributes *Outlook, Temperature, Humidity*, and *Wind*.

This last argument shows that Occam's razor will produce two different hypotheses from the same training examples when it is applied by two learners that perceive these examples in terms of different internal representations. On this basis we might be tempted to reject Occam's razor altogether. However, consider the following scenario that examines the question of which internal representations might arise from a process of evolution and natural selection. Imagine a population of artificial learning agents created by a simulated evolutionary process involving reproduction, mutation, and natural selection of these agents. Let us assume that this evolutionary process can alter the perceptual systems of these agents from generation to generation, thereby changing the internal attributes by which they perceive their world. For the sake of argument, let us also assume that the learning agents employ a fixed learning algorithm (say ID3) that cannot be altered by evolution. It is reasonable to assume that over time evolution will produce internal representation that make these agents increasingly successful within their environment. Assuming that the success of an agent depends highly on its ability to generalize accurately, we would therefore expect evolution to develop internal representations that work well with whatever learning algorithm and inductive bias is present. If the species of agents employs a learning algorithm whose inductive bias is Occam's razor, then we expect evolution to produce internal representations for which Occam's razor is a successful strategy. The essence of the argument here is that evolution will create internal representations that make the learning algorithm's inductive bias a self-fulfilling prophecy, simply because it can alter the representation easier than it can alter the learning algorithm.

For now, we leave the debate regarding Occam's razor. We will revisit it in Chapter 6, where we discuss the Minimum Description Length principle, a version of Occam's razor that can be interpreted within a Bayesian framework.

## 3.7 ISSUES IN DECISION TREE LEARNING

Practical issues in learning decision trees include determining how deeply to grow the decision tree, handling continuous attributes, choosing an appropriate attribute selection measure, handling training data with missing attribute values, handling attributes with differing costs, and improving computational efficiency. Below we discuss each of these issues and extensions to the basic ID3 algorithm that address them. ID3 has itself been extended to address most of these issues, with the resulting system renamed C4.5 (Quinlan 1993).

### 3.7.1 Avoiding Overfitting the Data

The algorithm described in Table 3.1 grows each branch of the tree just deeply enough to perfectly classify the training examples. While this is sometimes a

# CHAPTER 8

# INSTANCE-BASED LEARNING

In contrast to learning methods that construct a general, explicit description of the target function when training examples are provided, instance-based learning methods simply store the training examples. Generalizing beyond these examples is postponed until a new instance must be classified. Each time a new query instance is encountered, its relationship to the previously stored examples is examined in order to assign a target function value for the new instance. Instance-based learning includes nearest neighbor and locally weighted regression methods that assume instances can be represented as points in a Euclidean space. It also includes case-based reasoning methods that use more complex, symbolic representations for instances. Instance-based methods are sometimes referred to as "lazy" learning methods because they delay processing until a new instance must be classified. A key advantage of this kind of delayed, or lazy, learning is that instead of estimating the target function once for the entire instance space, these methods can estimate it locally and differently for each new instance to be classified.

## 8.1 INTRODUCTION

Instance-based learning methods such as nearest neighbor and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions. Learning in these algorithms consists of simply storing the presented training data. When a new query instance is encountered, a set of similar related instances is retrieved from memory and used to classify the

new query instance. One key difference between these approaches and the methods discussed in other chapters is that instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified. In fact, many techniques construct only a local approximation to the target function that applies in the neighborhood of the new query instance, and never construct an approximation designed to perform well over the entire instance space. This has significant advantages when the target function is very complex, but can still be described by a collection of less complex local approximations.

Instance-based methods can also use more complex, symbolic representations for instances. In case-based learning, instances are represented in this fashion and the process for identifying "neighboring" instances is elaborated accordingly. Case-based reasoning has been applied to tasks such as storing and reusing past experience at a help desk, reasoning about legal cases by referring to previous cases, and solving complex scheduling problems by reusing relevant portions of previously solved problems.

One disadvantage of instance-based approaches is that the cost of classifying new instances can be high. This is due to the fact that nearly all computation takes place at classification time rather than when the training examples are first encountered. Therefore, techniques for efficiently indexing training examples are a significant practical issue in reducing the computation required at query time. A second disadvantage to many instance-based approaches, especially nearest-neighbor approaches, is that they typically consider *all* attributes of the instances when attempting to retrieve similar training examples from memory. If the target concept depends on only a few of the many available attributes, then the instances that are truly most "similar" may well be a large distance apart.

In the next section we introduce the $k$-NEAREST NEIGHBOR learning algorithm, including several variants of this widely-used approach. The subsequent section discusses locally weighted regression, a learning method that constructs local approximations to the target function and that can be viewed as a generalization of $k$-NEAREST NEIGHBOR algorithms. We then describe radial basis function networks, which provide an interesting bridge between instance-based and neural network learning algorithms. The next section discusses case-based reasoning, an instance-based approach that employs symbolic representations and knowledge-based inference. This section includes an example application of case-based reasoning to a problem in engineering design. Finally, we discuss the fundamental differences in capabilities that distinguish lazy learning methods discussed in this chapter from eager learning methods discussed in the other chapters of this book.

## 8.2 $k$-NEAREST NEIGHBOR LEARNING

The most basic instance-based method is the $k$-NEAREST NEIGHBOR algorithm. This algorithm assumes all instances correspond to points in the $n$-dimensional space $\Re^n$. The nearest neighbors of an instance are defined in terms of the standard

Euclidean distance. More precisely, let an arbitrary instance $x$ be described by the feature vector

$$\langle a_1(x), a_2(x), \ldots a_n(x) \rangle$$

where $a_r(x)$ denotes the value of the $r$th attribute of instance $x$. Then the distance between two instances $x_i$ and $x_j$ is defined to be $d(x_i, x_j)$, where

$$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

In nearest-neighbor learning the target function may be either discrete-valued or real-valued. Let us first consider learning discrete-valued target functions of the form $f : \Re^n \rightarrow V$, where $V$ is the finite set $\{v_1, \ldots v_s\}$. The $k$-NEAREST NEIGHBOR algorithm for approximating a discrete-valued target function is given in Table 8.1. As shown there, the value $\hat{f}(x_q)$ returned by this algorithm as its estimate of $f(x_q)$ is just the most common value of $f$ among the $k$ training examples nearest to $x_q$. If we choose $k = 1$, then the 1-NEAREST NEIGHBOR algorithm assigns to $\hat{f}(x_q)$ the value $f(x_i)$ where $x_i$ is the training instance nearest to $x_q$. For larger values of $k$, the algorithm assigns the most common value among the $k$ nearest training examples.

Figure 8.1 illustrates the operation of the $k$-NEAREST NEIGHBOR algorithm for the case where the instances are points in a two-dimensional space and where the target function is boolean valued. The positive and negative training examples are shown by "+" and "−" respectively. A query point $x_q$ is shown as well. Note the 1-NEAREST NEIGHBOR algorithm classifies $x_q$ as a positive example in this figure, whereas the 5-NEAREST NEIGHBOR algorithm classifies it as a negative example.

What is the nature of the hypothesis space $H$ implicitly considered by the $k$-NEAREST NEIGHBOR algorithm? Note the $k$-NEAREST NEIGHBOR algorithm never forms an explicit general hypothesis $\hat{f}$ regarding the target function $f$. It simply computes the classification of each new query instance as needed. Nevertheless,

---

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance $x_q$ to be classified,
  - Let $x_1 \ldots x_k$ denote the $k$ instances from *training_examples* that are nearest to $x_q$
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

---

**TABLE 8.1**
The $k$-NEAREST NEIGHBOR algorithm for approximating a discrete-valued function $f : \Re^n \rightarrow V$.

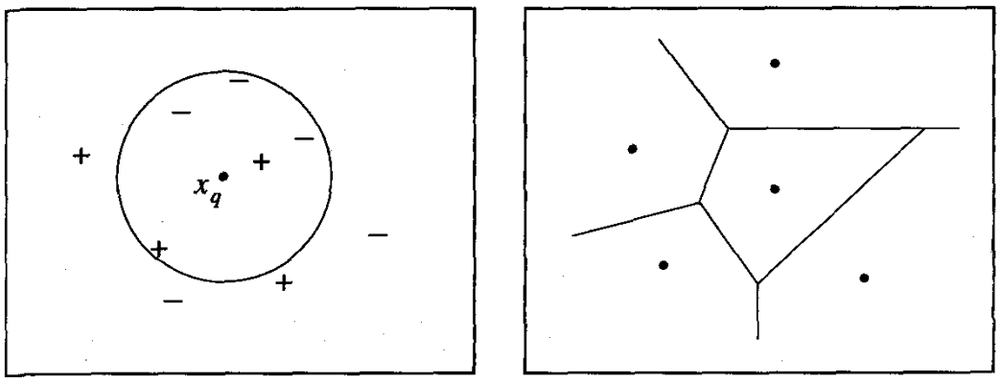



**FIGURE 8.1**
*k*-NEAREST NEIGHBOR. A set of positive and negative training examples is shown on the left, along with a query instance $x_q$ to be classified. The 1-NEAREST NEIGHBOR algorithm classifies $x_q$ positive, whereas 5-NEAREST NEIGHBOR classifies it as negative. On the right is the decision surface induced by the 1-NEAREST NEIGHBOR algorithm for a typical set of training examples. The convex polygon surrounding each training example indicates the region of instance space closest to that point (i.e., the instances for which the 1-NEAREST NEIGHBOR algorithm will assign the classification belonging to that training example).

we can still ask what the implicit general function is, or what classifications would be assigned if we were to hold the training examples constant and query the algorithm with every possible instance in $X$. The diagram on the right side of Figure 8.1 shows the shape of this decision surface induced by 1-NEAREST NEIGHBOR over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples. For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the *Voronoi diagram* of the set of training examples.

The *k*-NEAREST NEIGHBOR algorithm is easily adapted to approximating continuous-valued target functions. To accomplish this, we have the algorithm calculate the mean value of the $k$ nearest training examples rather than calculate their most common value. More precisely, to approximate a real-valued target function $f : \Re^n \rightarrow \Re$ we replace the final line of the above algorithm by the line

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k} \tag{8.1}$$

### 8.2.1 Distance-Weighted NEAREST NEIGHBOR Algorithm

One obvious refinement to the *k*-NEAREST NEIGHBOR algorithm is to weight the contribution of each of the $k$ neighbors according to their distance to the query point $x_q$, giving greater weight to closer neighbors. For example, in the algorithm of Table 8.1, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from $x_q$.

# CHAPTER 4

# ARTIFICIAL NEURAL NETWORKS

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Algorithms such as BACKPROPAGATION use gradient descent to tune network parameters to best fit a training set of input-output pairs. ANN learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot control strategies.

## 4.1 INTRODUCTION

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known. For example, the BACKPROPAGATION algorithm described in this chapter has proven surprisingly successful in many practical problems such as learning to recognize handwritten characters (LeCun et al. 1989), learning to recognize spoken words (Lang et al. 1990), and learning to recognize faces (Cottrell 1990). One survey of practical applications is provided by Rumelhart et al. (1994).

### 4.1.1 Biological Motivation

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

To develop a feel for this analogy, let us consider a few facts from neurobiology. The human brain, for example, is estimated to contain a densely interconnected network of approximately $10^{11}$ neurons, each connected, on average, to $10^4$ others. Neuron activity is typically excited or inhibited through connections to other neurons. The fastest neuron switching times are known to be on the order of $10^{-3}$ seconds—quite slow compared to computer switching speeds of $10^{-10}$ seconds. Yet humans are able to make surprisingly complex decisions, surprisingly quickly. For example, it requires approximately $10^{-1}$ seconds to visually recognize your mother. Notice the sequence of neuron firings that can take place during this $10^{-1}$-second interval cannot possibly be longer than a few hundred steps, given the switching speed of single neurons. This observation has led many to speculate that the information-processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons. One motivation for ANN systems is to capture this kind of highly parallel computation based on distributed representations. Most ANN software runs on sequential machines emulating distributed processes, although faster versions of the algorithms have also been implemented on highly parallel machines and on specialized hardware designed specifically for ANN applications.

While ANNs are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs, and many features of the ANNs we discuss here are known to be inconsistent with biological systems. For example, we consider here ANNs whose individual units output a single constant value, whereas biological neurons output a complex time series of spikes.

Historically, two groups of researchers have worked with artificial neural networks. One group has been motivated by the goal of using ANNs to study and model biological learning processes. A second group has been motivated by the goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes. Within this book our interest fits the latter group, and therefore we will not dwell further on biological modeling. For more information on attempts to model biological systems using ANNs, see, for example, Churchland and Sejnowski (1992); Zornetzer et al. (1994); Gabriel and Moore (1990).

## 4.2 NEURAL NETWORK REPRESENTATIONS

A prototypical example of ANN learning is provided by Pomerleau's (1993) system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving

at normal speeds on public highways. The input to the neural network is a $30 \times 32$ grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle. The network output is the direction in which the vehicle is steered. The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes. ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways (driving in the left lane of a divided public highway, with other vehicles present).

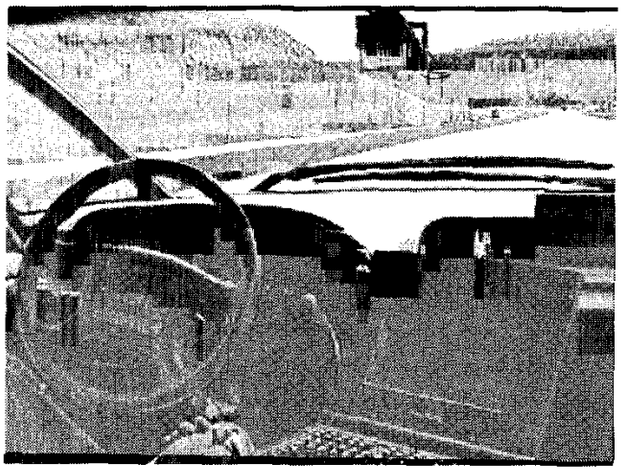
Figure 4.1 illustrates the neural network representation used in one version of the ALVINN system, and illustrates the kind of representation typical of many ANN systems. The network is shown on the left side of the figure, with the input camera image depicted below it. Each node (i.e., circle) in the network diagram corresponds to the output of a single network *unit*, and the lines entering the node from below are its inputs. As can be seen, there are four units that receive inputs directly from all of the $30 \times 32$ pixels in the image. These are called "hidden" units because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs. These hidden unit outputs are then used as inputs to a second layer of 30 "output" units. Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.

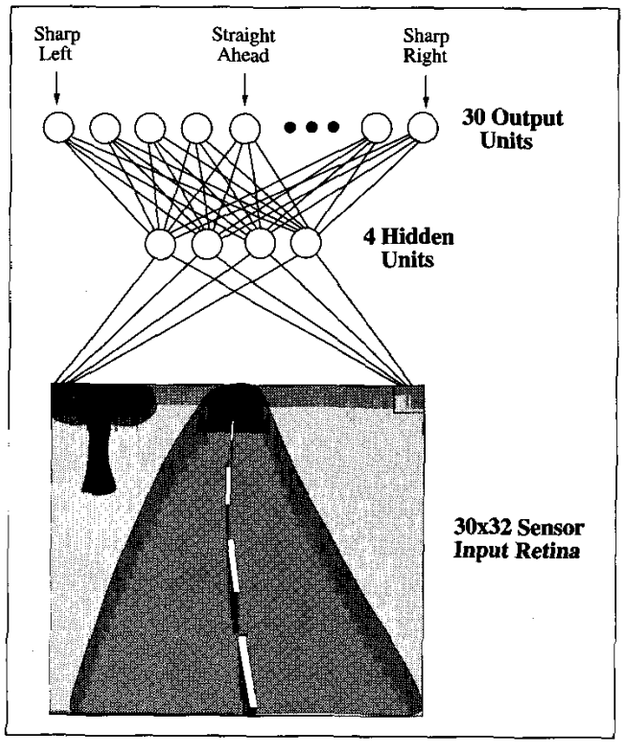
The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN. The large matrix of black and white boxes on the lower right depicts the weights from the $30 \times 32$ pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude. The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

The network structure of ALVINN is typical of many ANNs. Here the individual units are interconnected in layers that form a directed acyclic graph. In general, ANNs can be graphs with many types of structures—acyclic or cyclic, directed or undirected. This chapter will focus on the most common and practical ANN approaches, which are based on the BACKPROPAGATION algorithm. The BACKPROPAGATION algorithm assumes the network is a fixed structure that corresponds to a directed graph, possibly containing cycles. Learning corresponds to choosing a weight value for each edge in the graph. Although certain types of cycles are allowed, the vast majority of practical applications involve acyclic feed-forward networks, similar to the network structure used by ALVINN.

## 4.3 APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.





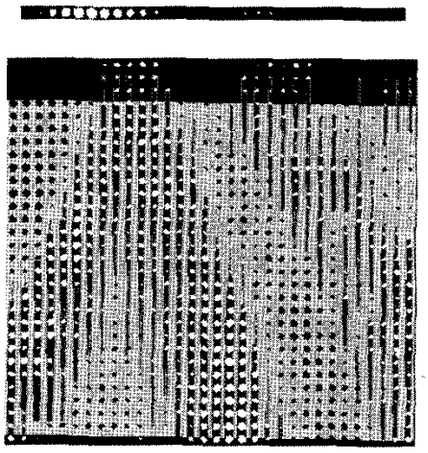



**FIGURE 4.1**
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 × 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

It is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks discussed in Chapter 3. In these cases ANN and decision tree learning often produce results of comparable accuracy. See Shavlik et al. (1991) and Weiss and Kapouleas (1989) for experimental comparisons of decision tree and ANN learning. The BACKPROPAGATION algorithm is the most commonly used ANN learning technique. It is appropriate for problems with the following characteristics:

- *Instances are represented by many attribute-value pairs.* The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- *The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.* For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction. We can also train a single network to output both the steering command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.
- *The training examples may contain errors.* ANN learning methods are quite robust to noise in the training data.
- *Long training times are acceptable.* Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- *Fast evaluation of the learned target function may be required.* Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.
- *The ability of humans to understand the learned target function is not important.* The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

The rest of this chapter is organized as follows: We first consider several alternative designs for the primitive units that make up artificial neural networks (perceptrons, linear units, and sigmoid units), along with learning algorithms for training single units. We then present the BACKPROPAGATION algorithm for training

multilayer networks of such units and consider several general issues such as the representational capabilities of ANNs, nature of the hypothesis space search, overfitting problems, and alternatives to the BACKPROPAGATION algorithm. A detailed example is also presented applying BACKPROPAGATION to face recognition, and directions are provided for the reader to obtain the data and code to experiment further with this application.

## 4.4 PERCEPTRONS

One type of ANN system is based on a unit called a *perceptron*, illustrated in Figure 4.2. A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and $-1$ otherwise. More precisely, given inputs $x_1$ through $x_n$, the output $o(x_1, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where each $w_i$ is a real-valued constant, or *weight*, that determines the contribution of input $x_i$ to the perceptron output. Notice the quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1 x_1 + \cdots + w_n x_n$ must surpass in order for the perceptron to output a 1.

To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} w_i x_i > 0$, or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

where

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$. Therefore, the space $H$ of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

### 4.4.1 Representational Power of Perceptrons

We can view the perceptron as representing a hyperplane decision surface in the $n$-dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a $-1$ for instances lying on the other side, as illustrated in Figure 4.3. The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called *linearly separable* sets of examples.

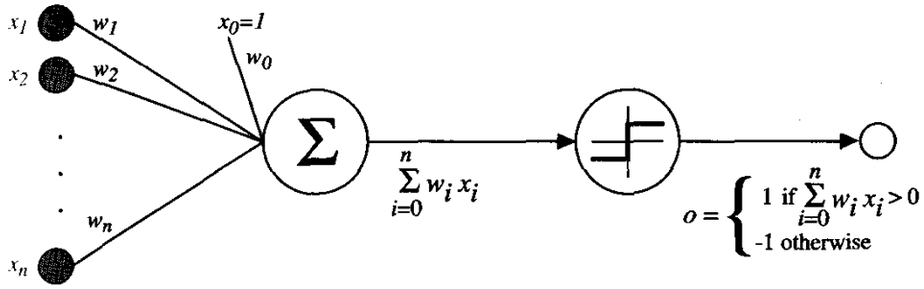



**FIGURE 4.2**
A perceptron.

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and $-1$ (false), then one way to use a two-input perceptron to implement the AND function is to set the weights $w_0 = -.8$, and $w_1 = w_2 = .5$. This perceptron can be made to represent the OR function instead by altering the threshold to $w_0 = -.3$. In fact, AND and OR can be viewed as special cases of $m$-of-$n$ functions: that is, functions where at least $m$ of the $n$ inputs to the perceptron must be true. The OR function corresponds to $m = 1$ and the AND function to $m = n$. Any $m$-of-$n$ function is easily represented using a perceptron by setting all input weights to the same value (e.g., 0.5) and then setting the threshold $w_0$ accordingly.

Perceptrons can represent all of the primitive boolean functions AND, OR, NAND ($\neg$ AND), and NOR ($\neg$ OR). Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_1 \neq x_2$. Note the set of linearly nonseparable training examples shown in Figure 4.3($b$) corresponds to this XOR function.

The ability of perceptrons to represent AND, OR, NAND, and NOR is important because *every* boolean function can be represented by some network of interconnected units based on these primitives. In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which

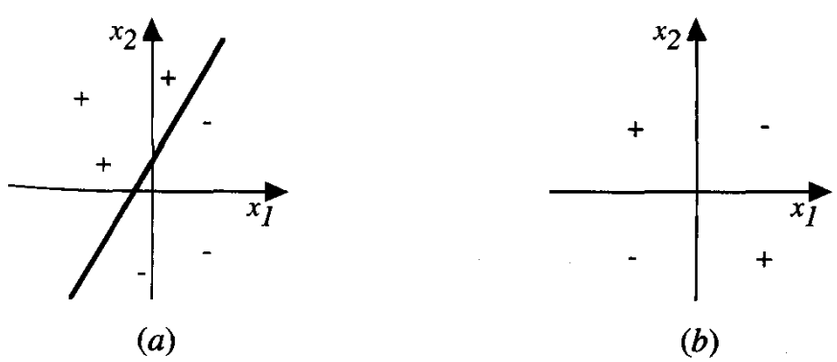



**FIGURE 4.3**
The decision surface represented by a two-input perceptron. ($a$) A set of training examples and the decision surface of a perceptron that classifies them correctly. ($b$) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). $x_1$ and $x_2$ are the perceptron inputs. Positive examples are indicated by "+", negative by "−".

the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage. One way is to represent the boolean function in disjunctive normal form (i.e., as the disjunction (OR) of a set of conjunctions (ANDs) of the inputs and their negations). Note that the input to an AND perceptron can be negated simply by changing the sign of the corresponding input weight.

Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

### 4.4.2 The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct $\pm 1$ output for each of the given training examples.

Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule (a variant of the LMS rule used in Chapter 1 for learning evaluation functions). These two algorithms are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions. They are important to ANNs because they provide the basis for learning networks of many units.

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the *perceptron training rule*, which revises the weight $w_i$ associated with input $x_i$ according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Here $t$ is the target output for the current training example, $o$ is the output generated by the perceptron, and $\eta$ is a positive constant called the *learning rate*. The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

Why should this update rule converge toward successful weight values? To get an intuitive feel, consider some specific cases. Suppose the training example is correctly classified already by the perceptron. In this case, $(t - o)$ is zero, making $\Delta w_i$ zero, so that no weights are updated. Suppose the perceptron outputs a $-1$, when the target output is $+1$. To make the perceptron output a $+1$ instead of $-1$ in this case, the weights must be altered to increase the value of $\vec{w} \cdot \vec{x}$. For example, if $x_i > 0$, then increasing $w_i$ will bring the perceptron closer to correctly classifying

this example. Notice the training rule will increase $w_i$ in this case, because $(t-o)$, $\eta$, and $x_i$ are all positive. For example, if $x_i = .8$, $\eta = 0.1$, $t = 1$, and $o = -1$, then the weight update will be $\Delta w_i = \eta(t-o)x_i = 0.1(1-(-1))0.8 = 0.16$. On the other hand, if $t = -1$ and $o = 1$, then weights associated with positive $x_i$ will be decreased rather than increased.

In fact, the above learning procedure can be proven to converge within a finite number of applications of the perceptron training rule to a weight vector that correctly classifies all training examples, *provided the training examples are linearly separable* and provided a sufficiently small $\eta$ is used (see Minsky and Papert 1969). If the data are not linearly separable, convergence is not assured.

### 4.4.3 Gradient Descent and the Delta Rule

Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable. A second training rule, called the *delta rule*, is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.

The key idea behind the delta rule is to use *gradient descent* to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples. This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

The delta training rule is best understood by considering the task of training an *unthresholded* perceptron; that is, a *linear unit* for which the output $o$ is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \tag{4.1}$$

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the *training error* of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \tag{4.2}$$

where $D$ is the set of training examples, $t_d$ is the target output for training example $d$, and $o_d$ is the output of the linear unit for training example $d$. By this definition, $E(\vec{w})$ is simply half the squared difference between the target output $t_d$ and the linear unit output $o_d$, summed over all training examples. Here we characterize $E$ as a function of $\vec{w}$ because the linear unit output $o$ depends on this weight vector. Of course $E$ also depends on the particular set of training examples, but

we assume these are fixed during training, so we do not bother to write $E$ as an explicit function of these. Chapter 6 provides a Bayesian justification for choosing this particular definition of $E$. In particular, there we show that under certain conditions the hypothesis that minimizes $E$ is also the most probable hypothesis in $H$ given the training data.

#### 4.4.3.1 VISUALIZING THE HYPOTHESIS SPACE

To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated $E$ values, as illustrated in Figure 4.4. Here the axes $w_0$ and $w_1$ represent possible values for the two weights of a simple linear unit. The $w_0, w_1$ plane therefore represents the entire hypothesis space. The vertical axis indicates the error $E$ relative to some fixed set of training examples. The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space (we desire a hypothesis with minimum error). Given the way in which we chose to define $E$, for linear units this error surface must always be parabolic with a single global minimum. The specific parabola will depend, of course, on the particular set of training examples.

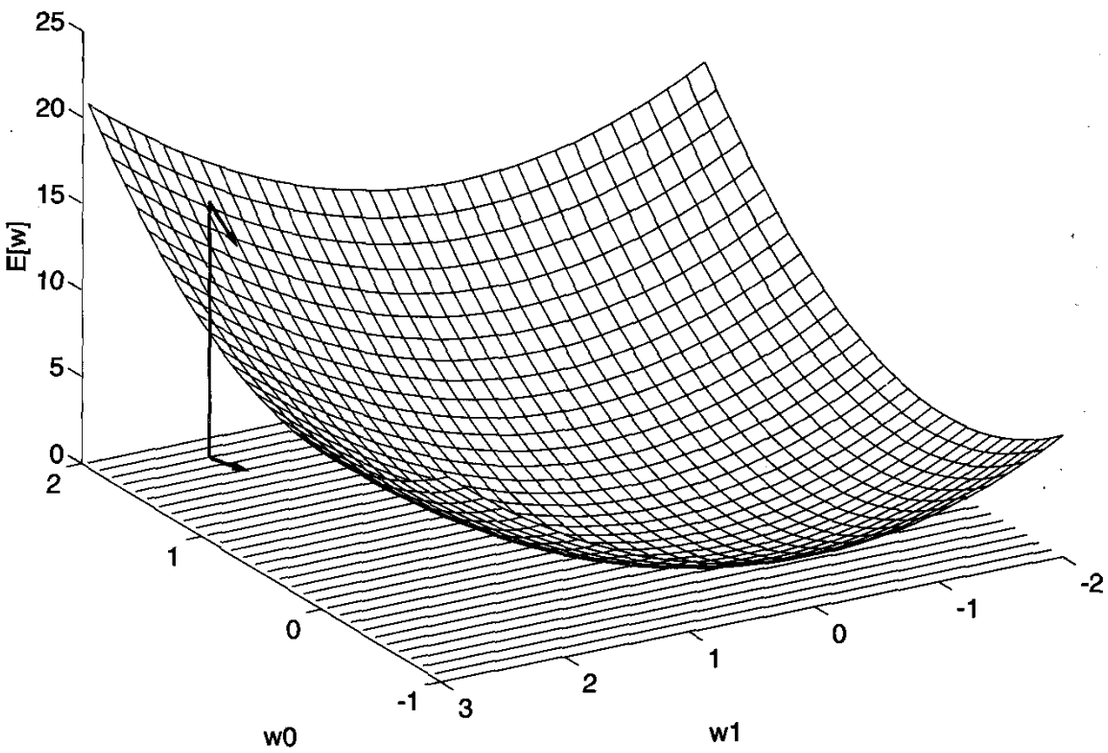



**FIGURE 4.4**
Error of different hypotheses. For a linear unit with two weights, the hypothesis space $H$ is the $w_0, w_1$ plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the $w_0, w_1$ plane producing steepest descent along the error surface.

Gradient descent search determines a weight vector that minimizes $E$ by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in Figure 4.4. This process continues until the global minimum error is reached.

#### 4.4.3.2 DERIVATION OF THE GRADIENT DESCENT RULE

How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of $E$ with respect to each component of the vector $\vec{w}$. This vector derivative is called the *gradient* of $E$ with respect to $\vec{w}$, written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \tag{4.3}$$

Notice $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of $E$ with respect to each of the $w_i$. *When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in* $E$. The negative of this vector therefore gives the direction of steepest decrease. For example, the arrow in Figure 4.4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the $w_0$, $w_1$ plane.

Since the gradient specifies the direction of steepest increase of $E$, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

where

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \tag{4.4}$$

Here $\eta$ is a positive constant called the learning rate, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that *decreases* $E$. This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \tag{4.5}$$

which makes it clear that steepest descent is achieved by altering each component $w_i$ of $\vec{w}$ in proportion to $\frac{\partial E}{\partial w_i}$.

To construct a practical algorithm for iteratively updating weights according to Equation (4.5), we need an efficient way of calculating the gradient at each step. Fortunately, this is not difficult. The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the

gradient can be obtained by differentiating $E$ from Equation (4.2), as

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id}) \qquad (4.6)
\end{aligned}$$

where $x_{id}$ denotes the single input component $x_i$ for training example $d$. We now have an equation that gives $\frac{\partial E}{\partial w_i}$ in terms of the linear unit inputs $x_{id}$, outputs $O_d$, and target values $t_d$ associated with the training examples. Substituting Equation (4.6) into Equation (4.5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d\in D}(t_d - o_d)\, x_{id} \qquad (4.7)$$

To summarize, the gradient descent algorithm for training linear units is as follows: Pick an initial random weight vector. Apply the linear unit to all training examples, then compute $\Delta w_i$ for each weight according to Equation (4.7). Update each weight $w_i$ by adding $\Delta w_i$, then repeat this process. This algorithm is given in Table 4.1. Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate $\eta$ is used. If $\eta$ is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to gradually reduce the value of $\eta$ as the number of gradient descent steps grows.

#### 4.4.3.3 STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever (1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and (2) the error can be differentiated with respect to these hypothesis parameters. The key practical difficulties in applying gradient descent are (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and (2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

GRADIENT-DESCENT(*training_examples*, $\eta$)

*Each training example is a pair of the form* $\langle \vec{x}, t \rangle$, *where* $\vec{x}$ *is the vector of input values, and* $t$ *is the target output value.* $\eta$ *is the learning rate (e.g., .05).*

- Initialize each $w_i$ to some small random value
- Until the termination condition is met, Do
  - Initialize each $\Delta w_i$ to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output $o$
    - For each linear unit weight $w_i$, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \tag{T4.1}$$

  - For each linear unit weight $w_i$, Do

$$w_i \leftarrow w_i + \Delta w_i \tag{T4.2}$$

**TABLE 4.1**
GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t-o)x_i$.

One common variation on gradient descent intended to alleviate these difficulties is called *incremental gradient descent*, or alternatively *stochastic gradient descent*. Whereas the gradient descent training rule presented in Equation (4.7) computes weight updates after summing over *all* the training examples in $D$, the idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for *each* individual example. The modified training rule is like the training rule given by Equation (4.7) except that as we iterate through each training example we update the weight according to

$$\Delta w_i = \eta(t - o)\ x_i \tag{4.10}$$

where $t, o$, and $x_i$ are the target value, unit output, and $i$th input for the training example in question. To modify the gradient descent algorithm of Table 4.1 to implement this stochastic approximation, Equation (T4.2) is simply deleted and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t-o)\ x_i$. One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ defined for each individual training example $d$ as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \tag{4.11}$$

where $t_d$ and $o_d$ are the target value and the unit output value for training example $d$. Stochastic gradient descent iterates over the training examples $d$ in $D$, at each iteration altering the weights according to the gradient with respect to $E_d(\vec{w})$. The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function $E(\vec{w})$. By making the value of $\eta$ (the gradient

descent step size) sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely. The key differences between standard gradient descent and stochastic gradient descent are:

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
- In cases where there are multiple local minima with respect to $E(\vec{w})$, stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search.

Both stochastic and standard gradient descent methods are commonly used in practice.

The training rule in Equation (4.10) is known as the *delta rule*, or sometimes the LMS (least-mean-square) rule, Adaline rule, or Widrow-Hoff rule (after its inventors). In Chapter 1 we referred to it as the LMS weight-update rule when describing its use for learning an evaluation function for game playing. Notice the delta rule in Equation (4.10) is similar to the perceptron training rule in Equation (4.4.2). In fact, the two expressions appear to be identical. However, the rules are different because in the delta rule $o$ refers to the linear unit output $o(\vec{x}) = \vec{w} \cdot \vec{x}$, whereas for the perceptron rule $o$ refers to the thresholded output $o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$.

Although we have presented the delta rule as a method for learning weights for unthresholded linear units, it can easily be used to train thresholded perceptron units, as well. Suppose that $o = \vec{w} \cdot \vec{x}$ is the unthresholded linear unit output as above, and $o' = sgn(\vec{w} \cdot \vec{x})$ is the result of thresholding $o$ as in the perceptron. Now if we wish to train a perceptron to fit training examples with target values of $\pm 1$ for $o'$, we can use these same target values and examples to train $o$ instead, using the delta rule. Clearly, if the unthresholded output $o$ can be trained to fit these values perfectly, then the threshold output $o'$ will fit them as well (because $sgn(1) = 1$, and $sgn(-1) = -1$). Even when the target values cannot be fit perfectly, the thresholded $o'$ value will correctly fit the $\pm 1$ target value whenever the linear unit output $o$ has the correct sign. Notice, however, that while this procedure will learn weights that minimize the error in the linear unit output $o$, these weights will not necessarily minimize the number of training examples misclassified by the thresholded output $o'$.

### 4.4.4 Remarks

We have considered two similar algorithms for iteratively learning perceptron weights. The key difference between these algorithms is that the perceptron train-

ing rule updates weights based on the error in the *thresholded* perceptron output, whereas the delta rule updates weights based on the error in the *unthresholded* linear combination of inputs.

The difference between these two training rules is reflected in different convergence properties. The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, *provided the training examples are linearly separable*. The delta rule converges only asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges *regardless of whether the training data are linearly separable*. A detailed presentation of the convergence proofs can be found in Hertz et al. (1991).

A third possible algorithm for learning the weight vector is linear programming. Linear programming is a general, efficient method for solving sets of linear inequalities. Notice each training example corresponds to an inequality of the form $\vec{w} \cdot \vec{x} > 0$ or $\vec{w} \cdot \vec{x} \leq 0$, and their solution is the desired weight vector. Unfortunately, this approach yields a solution only when the training examples are linearly separable; however, Duda and Hart (1973, p. 168) suggest a more subtle formulation that accommodates the nonseparable case. In any case, the approach of linear programming does not scale to training multilayer networks, which is our primary concern. In contrast, the gradient descent approach, on which the delta rule is based, can be easily extended to multilayer networks, as shown in the following section.

## 4.5 MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

As noted in Section 4.4.1, single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the BACKPROPAGATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces. For example, a typical multilayer network and decision surface is depicted in Figure 4.5. Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h_d" (i.e., "hid," "had," "head," "hood," etc.). The input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily visualize the decision surface over the two-dimensional instance space. As shown in the figure, it is possible for the multilayer network to represent highly nonlinear decision surfaces that are much more expressive than the linear decision surfaces of single units shown earlier in Figure 4.3.

This section discusses how to learn such multilayer networks using a gradient descent algorithm similar to that discussed in the previous section.

### 4.5.1 A Differentiable Threshold Unit

What type of unit shall we use as the basis for constructing multilayer networks? At first we might be tempted to choose the linear units discussed in the previous

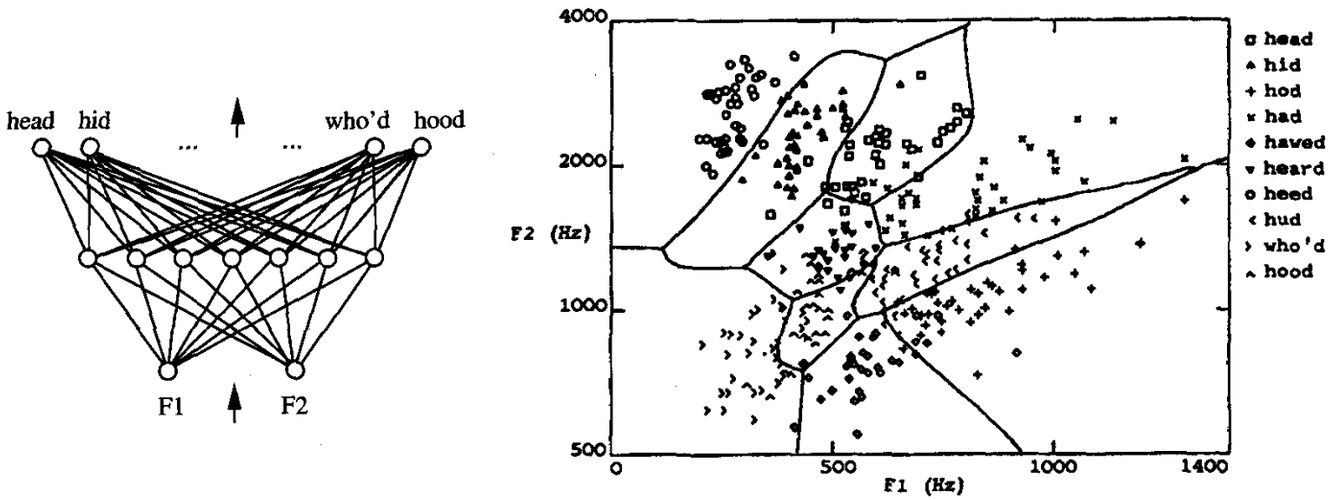



**FIGURE 4.5**
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest. The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network. (Reprinted by permission from Haung and Lippmann (1988).)

section, for which we have already derived a gradient descent learning rule. However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the *sigmoid unit*—a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

The sigmoid unit is illustrated in Figure 4.6. Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a

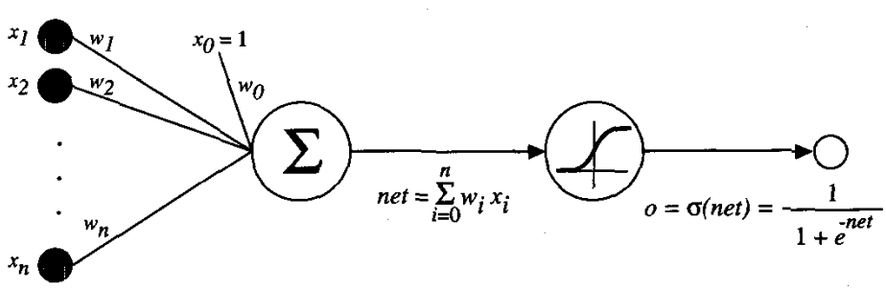



**FIGURE 4.6**
The sigmoid threshold unit.

continuous function of its input. More precisely, the sigmoid unit computes its output $o$ as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}} \tag{4.12}$$

$\sigma$ is often called the sigmoid function or, alternatively, the logistic function. Note its output ranges between 0 and 1, increasing monotonically with its input (see the threshold function plot in Figure 4.6.). Because it maps a very large input domain to a small range of outputs, it is often referred to as the *squashing function* of the unit. The sigmoid function has the useful property that its derivative is easily expressed in terms of its output [in particular, $\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$]. As we shall see, the gradient descent learning rule makes use of this derivative. Other differentiable functions with easily calculated derivatives are sometimes used in place of $\sigma$. For example, the term $e^{-y}$ in the sigmoid function definition is sometimes replaced by $e^{-k \cdot y}$ where $k$ is some positive constant that determines the steepness of the threshold. The function $tanh$ is also sometimes used in place of the sigmoid function (see Exercise 4.8).

### 4.5.2 The BACKPROPAGATION Algorithm

The BACKPROPAGATION algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs. This section presents the BACKPROPAGATION algorithm, and the following section gives the derivation for the gradient descent weight update rule used by BACKPROPAGATION.

Because we are considering networks with multiple output units rather than single units as before, we begin by redefining $E$ to sum the errors over all of the network output units

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \tag{4.13}$$

where $outputs$ is the set of output units in the network, and $t_{kd}$ and $o_{kd}$ are the target and output values associated with the $k$th output unit and training example $d$.

The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by all possible weight values for all the units in the network. The situation can be visualized in terms of an error surface similar to that shown for linear units in Figure 4.4. The error in that diagram is replaced by our new definition of $E$, and the other dimensions of the space correspond now to all of the weights associated with all of the units in the network. As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize $E$.

BACKPROPAGATION($training_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

*Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.*

*$\eta$ is the learning rate (e.g., .05). $n_{in}$ is the number of network inputs, $n_{hidden}$ the number of units in the hidden layer, and $n_{out}$ the number of output units.*

*The input from unit i into unit j is denoted $x_{ji}$, and the weight from unit i to unit j is denoted $w_{ji}$.*

- Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in *training_examples*, Do

    *Propagate the input forward through the network:*

    1. Input the instance $\vec{x}$ to the network and compute the output $o_u$ of every unit $u$ in the network.

    *Propagate the errors backward through the network:*

    2. For each network output unit $k$, calculate its error term $\delta_k$

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

    3. For each hidden unit $h$, calculate its error term $\delta_h$

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

    4. Update each network weight $w_{ji}$

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

    where

$$\Delta w_{ji} = \eta\, \delta_j\, x_{ji} \tag{T4.5}$$

**TABLE 4.2**
The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks containing two layers of sigmoid units.

One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface shown in Figure 4.4. Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error. Despite this obstacle, in practice BACKPROPAGATION has been found to produce excellent results in many real-world applications.

The BACKPROPAGATION algorithm is presented in Table 4.2. The algorithm as described here applies to layered feedforward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of BACKPROPAGATION. The notation used here is the same as that used in earlier sections, with the following extensions:

- An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
- $x_{ji}$ denotes the input from node $i$ to unit $j$, and $w_{ji}$ denotes the corresponding weight.
- $\delta_n$ denotes the error term associated with unit $n$. It plays a role analogous to the quantity $(t - o)$ in our earlier discussion of the delta training rule. As we shall see later, $\delta_n = -\frac{\partial E}{\partial net_n}$.

Notice the algorithm in Table 4.2 begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. Given this fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples. For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

The gradient descent weight-update rule (Equation [T4.5] in Table 4.2) is similar to the delta training rule (Equation [4.10]). Like the delta rule, it updates each weight in proportion to the learning rate $\eta$, the input value $x_{ji}$ to which the weight is applied, and the error in the output of the unit. The only difference is that the error $(t - o)$ in the delta rule is replaced by a more complex error term, $\delta_j$. The exact form of $\delta_j$ follows from the derivation of the weight-tuning rule given in Section 4.5.3. To understand it intuitively, first consider how $\delta_k$ is computed for each network *output* unit $k$ (Equation [T4.3] in the algorithm). $\delta_k$ is simply the familiar $(t_k - o_k)$ from the delta rule, multiplied by the factor $o_k(1 - o_k)$, which is the derivative of the sigmoid squashing function. The $\delta_h$ value for each *hidden* unit $h$ has a similar form (Equation [T4.4] in the algorithm). However, since training examples provide target values $t_k$ *only* for network outputs, no target values are directly available to indicate the error of hidden units' values. Instead, the error term for hidden unit $h$ is calculated by summing the error terms $\delta_k$ for each output unit influenced by $h$, weighting each of the $\delta_k$'s by $w_{kh}$, the weight from hidden unit $h$ to output unit $k$. This weight characterizes the degree to which hidden unit $h$ is "responsible for" the error in output unit $k$.

The algorithm in Table 4.2 updates weights incrementally, following the presentation of each training example. This corresponds to a stochastic approximation to gradient descent. To obtain the true gradient of $E$ one would sum the $\delta_j x_{ji}$ values over all training examples before altering weight values.

The weight-update loop in BACKPROPAGATION may be iterated thousands of times in a typical application. A variety of termination conditions can be used to halt the procedure. One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below some threshold, or once the error on a separate validation set of examples meets some

criterion. The choice of termination criterion is an important one, because too few iterations can fail to reduce error sufficiently, and too many can lead to overfitting the training data. This issue is discussed in greater detail in Section 4.6.5.

#### 4.5.2.1 ADDING MOMENTUM

Because BACKPROPAGATION is such a widely used algorithm, many variations have been developed. Perhaps the most common is to alter the weight-update rule in Equation (T4.5) in the algorithm by making the weight update on the $n$th iteration depend partially on the update that occurred during the $(n-1)$th iteration, as follows:

$$\Delta w_{ji}(n) = \eta\, \delta_j\, x_{ji} + \alpha \Delta w_{ji}(n-1) \tag{4.18}$$

Here $\Delta w_{ji}(n)$ is the weight update performed during the $n$th iteration through the main loop of the algorithm, and $0 \leq \alpha < 1$ is a constant called the *momentum*. Notice the first term on the right of this equation is just the weight-update rule of Equation (T4.5) in the BACKPROPAGATION algorithm. The second term on the right is new and is called the momentum term. To see the effect of this momentum term, consider that the gradient descent search trajectory is analogous to that of a (momentumless) ball rolling down the error surface. The effect of $\alpha$ is to add momentum that tends to keep the ball rolling in the same direction from one iteration to the next. This can sometimes have the effect of keeping the ball rolling through small local minima in the error surface, or along flat regions in the surface where the ball would stop if there were no momentum. It also has the effect of gradually increasing the step size of the search in regions where the gradient is unchanging, thereby speeding convergence.

#### 4.5.2.2 LEARNING IN ARBITRARY ACYCLIC NETWORKS

The definition of BACKPROPAGATION presented in Table 4.2 applies only to two-layer networks. However, the algorithm given there easily generalizes to feedforward networks of arbitrary depth. The weight update rule seen in Equation (T4.5) is retained, and the only change is to the procedure for computing $\delta$ values. In general, the $\delta_r$ value for a unit $r$ in layer $m$ is computed from the $\delta$ values at the next deeper layer $m+1$ according to

$$\delta_r = o_r\,(1 - o_r) \sum_{s \in layer\, m+1} w_{sr}\, \delta_s \tag{4.19}$$

Notice this is identical to Step 3 in the algorithm of Table 4.2, so all we are really saying here is that this step may be repeated for any number of hidden layers in the network.

It is equally straightforward to generalize the algorithm to any directed acyclic graph, regardless of whether the network units are arranged in uniform layers as we have assumed up to now. In the case that they are not, the rule for calculating $\delta$ for any internal unit (i.e., any unit that is not an output) is

$$\delta_r = o_r\,(1 - o_r) \sum_{s \in Downstream(r)} w_{sr}\, \delta_s \tag{4.20}$$

residuals. This is unlikely to converge to the exact function, so the algorithm simply runs for a fixed number of training epochs (an epoch is one complete run through the training data). Alternatively, one could run the algorithm until a bound on the sum of squared residuals is reached.

## 7.3 Support vector machines

Linearly separable data admits infinitely many decision boundaries that separate the classes, but intuitively some of these are better than others. For example, the left and middle decision boundaries in Figure 7.5 seem to be unnecessarily close to some of the positives; while the one on the right leaves a bit more space on either side, it doesn't seem particularly good either. To make this a bit more precise, recall that in Section 2.2 we defined the ☞*margin* of an example assigned by a scoring classifier as $c(x)\hat{s}(x)$, where $c(x)$ is $+1$ for positive examples and $-1$ for negative examples and $\hat{s}(x)$ is the score of example $x$. If we take $\hat{s}(\mathbf{x}) = \mathbf{w}\cdot\mathbf{x} - t$, then a true positive $\mathbf{x}_i$ has margin $\mathbf{w}\cdot\mathbf{x}_i - t > 0$ and a true negative $\mathbf{x}_j$ has margin $-(\mathbf{w}\cdot\mathbf{x}_j - t) > 0$. For a given training set and decision boundary, let $m^{\oplus}$ be the smallest margin of any positive, and $m^{\ominus}$ the smallest margin of any negative, then we want the sum of these to be as large as possible. This sum is independent of the decision threshold $t$, as long as we keep the nearest positives and negatives at the right sides of the decision boundary, and so we re-adjust $t$ such that $m^{\oplus}$ and $m^{\ominus}$ become equal. Figure 7.7 depicts this graphically in a two-dimensional instance space. The training examples nearest to the decision boundary are called *support vectors*: as we shall see, the decision boundary of a support vector machine (SVM) is defined as a linear combination of the support vectors.

The *margin* is thus defined as $m/||\mathbf{w}||$, where $m$ is the distance between the decision boundary and the nearest training instances (at least one of each class) as

**Algorithm 7.3:** PerceptronRegression($D$, $T$) – train a perceptron for regression.

**Input** : labelled training data $D$ in homogeneous coordinates;
maximum number of training epochs $T$.

**Output** : weight vector $\mathbf{w}$ defining function approximator $\hat{y} = \mathbf{w}\cdot\mathbf{x}$.

```
w ←0; t ←0;
while t < T do
    for i = 1 to |D| do
        w←w + (y_i − ŷ_i)^2 x_i;
    end
    t ← t + 1;
end
```

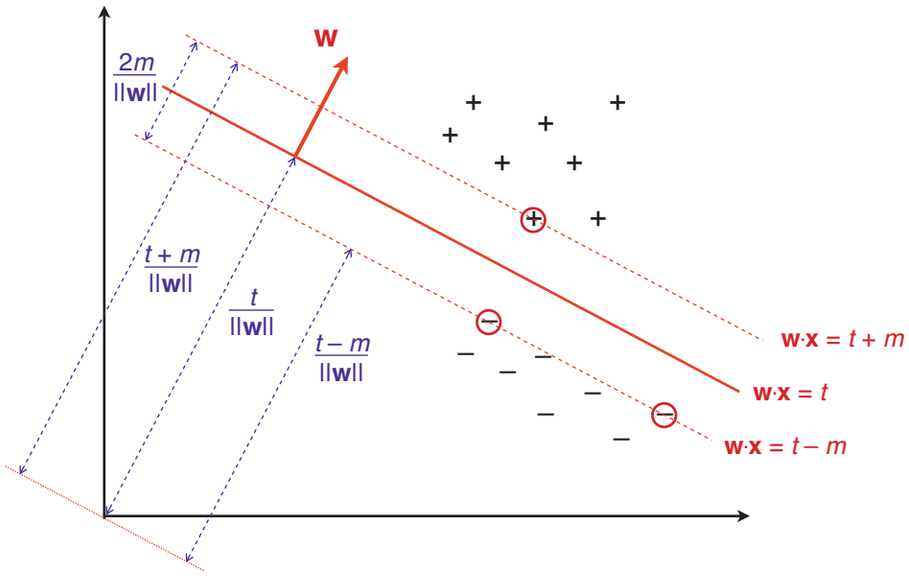



**Figure 7.7.** The geometry of a support vector classifier. The circled data points are the support vectors, which are the training examples nearest to the decision boundary. The support vector machine finds the decision boundary that maximises the margin $m/||\mathbf{w}||$.

measured along $\mathbf{w}$. Since we are free to rescale $t$, $||\mathbf{w}||$ and $m$, it is customary to choose $m = 1$. Maximising the margin then corresponds to minimising $||\mathbf{w}||$ or, more conveniently, $\frac{1}{2}||\mathbf{w}||^2$, provided of course that none of the training points fall inside the margin. This leads to a quadratic, constrained optimisation problem:

$$\mathbf{w}^*, t^* = \underset{\mathbf{w},t}{\operatorname{argmin}} \frac{1}{2}||\mathbf{w}||^2 \qquad \text{subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i - t) \geq 1, 1 \leq i \leq n$$

We will approach this using the method of Lagrange multipliers (see Background 7.3). Adding the constraints with multipliers $\alpha_i$ for each training example gives the Lagrange function

$$\begin{aligned}
\Lambda(\mathbf{w}, t, \alpha_1, \ldots, \alpha_n) &= \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i (y_i(\mathbf{w}\cdot\mathbf{x}_i - t) - 1) \\
&= \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{n} \alpha_i y_i (\mathbf{w}\cdot\mathbf{x}_i) + \sum_{i=1}^{n} \alpha_i y_i t + \sum_{i=1}^{n} \alpha_i \\
&= \frac{1}{2}\mathbf{w}\cdot\mathbf{w} - \mathbf{w}\cdot\left(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i\right) + t\left(\sum_{i=1}^{n} \alpha_i y_i\right) + \sum_{i=1}^{n} \alpha_i
\end{aligned}$$

While this looks like a formidable formula, some further analysis will allow us to derive the simpler dual form of the Lagrange function.

By taking the partial derivative of the Lagrange function with respect to $t$ and setting it to 0 we find that for the optimal threshold $t$ we have $\sum_{i=1}^{n} \alpha_i y_i = 0$. Similarly, by

Optimisation is a broad term denoting the problem of finding the best item or value among a set of alternatives. We have already seen a very simple, unconstrained form of optimisation in Example 7.1 on p.197, where we found the values of $a$ and $b$ minimising the sum of squared residuals $f(a,b) = \sum_{i=1}^{n}(w_i - (a + bh_i))^2$; this can be denoted as

$$a^*, b^* = \underset{a,b}{\arg\min}\, f(a,b)$$

$f$ is called the *objective function*; it can be linear, quadratic (as in this case), or more complex. We found the minimum of $f$ by setting the partial derivatives of $f$ with respect to $a$ and $b$ to 0, and solving for $a$ and $b$; the vector of these partial derivatives is called the *gradient* and denoted $\nabla f$, so a succinct way of defining the unconstrained optimisation problem is: find $a$ and $b$ such that $\nabla f(a,b) = \mathbf{0}$. In this particular case the objective function is *convex*, which essentially means that there is a unique global minimum. This is, however, not always the case.

A *constrained optimisation* problem is one where the alternatives are subject to constraints, for instance

$$a^*, b^* = \underset{a,b}{\arg\min}\, f(a,b) \qquad \text{subject to } g(a,b) = c$$

If the relationship expressed by the constraint is linear, say $a - b = 0$, we can of course eliminate one of the variables and solve the simpler, unconstrained problem. However, this may not be possible if the constraints are non-linear. *Lagrange multipliers* are a powerful way of dealing with the general case. We form the Lagrange function defined by

$$\Lambda(a,b,\lambda) = f(a,b) - \lambda(g(a,b) - c)$$

where $\lambda$ is the Lagrange multiplier, and solve the unconstrained problem $\nabla\Lambda(a,b,\lambda) = \mathbf{0}$. Since $\nabla_{a,b}\Lambda(a,b,\lambda) = \nabla f(a,b) - \lambda\nabla g(a,b)$ and $\nabla_\lambda \Lambda(a,b,\lambda) = g(a,b) - c$, this is a succinct way of requiring *(i)* that the gradients of $f$ and $g$ point in the same direction, and *(ii)* that the constraint is satisfied. We can include multiple equality constraints and also inequality constraints, each with their own Lagrange multiplier.

From the Lagrange function it is possible to derive a *dual* optimisation problem where we find the optimal values of the Lagrange multipliers. In general, the solution to the dual problem is only a lower bound on the solution to the *primal* problem, but under a set of conditions known as the *Karush–Kuhn–Tucker conditions* (*KKT*) the two solutions become equal. The quadratic optimisation problem posed by support vector machines is usually solved in its dual form.

**Background 7.3.** Basic concepts and terminology in mathematical optimisation.

taking the partial derivative of the Lagrange function with respect to $\mathbf{w}$ we see that the Lagrange multipliers define the weight vector as a linear combination of the training examples:

$$\frac{\partial}{\partial \mathbf{w}}\Lambda(\mathbf{w}, t, \alpha_1, \ldots, \alpha_n) = \frac{\partial}{\partial \mathbf{w}}\frac{1}{2}\mathbf{w}\cdot\mathbf{w} - \frac{\partial}{\partial \mathbf{w}}\mathbf{w}\cdot\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right) = \mathbf{w} - \sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i$$

Since this partial derivative is 0 for an optimal weight vector we conclude $\mathbf{w} = \sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i$ – the same expression as we derived for the perceptron in Equation 7.9 on p.209. For the perceptron, the instance weights $\alpha_i$ are non-negative integers denoting the number of times an example has been misclassified in training. For a support vector machine, the $\alpha_i$ are non-negative reals. What they have in common is that, if $\alpha_i = 0$ for a particular example $\mathbf{x}_i$, that example could be removed from the training set without affecting the learned decision boundary. In the case of support vector machines this means that $\alpha_i > 0$ only for the support vectors: the training examples nearest to the decision boundary.

Now, by plugging the expressions $\sum_{i=1}^{n}\alpha_i y_i = 0$ and $\mathbf{w} = \sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i$ back into the Lagrangian we are able to eliminate $\mathbf{w}$ and $t$, and hence obtain the dual optimisation problem, which is entirely formulated in terms of the Lagrange multipliers:

$$\begin{aligned}\Lambda(\alpha_1, \ldots, \alpha_n) &= -\frac{1}{2}\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right)\cdot\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right) + \sum_{i=1}^{n}\alpha_i \\ &= -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \mathbf{x}_i\cdot\mathbf{x}_j + \sum_{i=1}^{n}\alpha_i\end{aligned}$$

The dual problem is to maximise this function under positivity constraints and one equality constraint:

$$\alpha_1^*, \ldots, \alpha_n^* = \underset{\alpha_1,\ldots,\alpha_n}{\arg\max} -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \mathbf{x}_i\cdot\mathbf{x}_j + \sum_{i=1}^{n}\alpha_i$$
$$\text{subject to } \alpha_i \geq 0, 1 \leq i \leq n \text{ and } \sum_{i=1}^{n}\alpha_i y_i = 0$$

The dual form of the optimisation problem for support vector machines illustrates two important points. First, it shows that searching for the maximum-margin decision boundary is equivalent to searching for the support vectors: they are the training examples with non-zero Lagrange multipliers, and through $\mathbf{w} = \sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i$ they completely determine the decision boundary. Secondly, it shows that the optimisation problem is entirely defined by pairwise dot products between training instances: the entries of the Gram matrix. As we shall see in Section 7.5, this paves the way for a powerful adaptation of support vector machines that allows them to learn non-linear decision boundaries.

The following example makes these issues a bit more concrete by showing detailed calculations on some toy data.

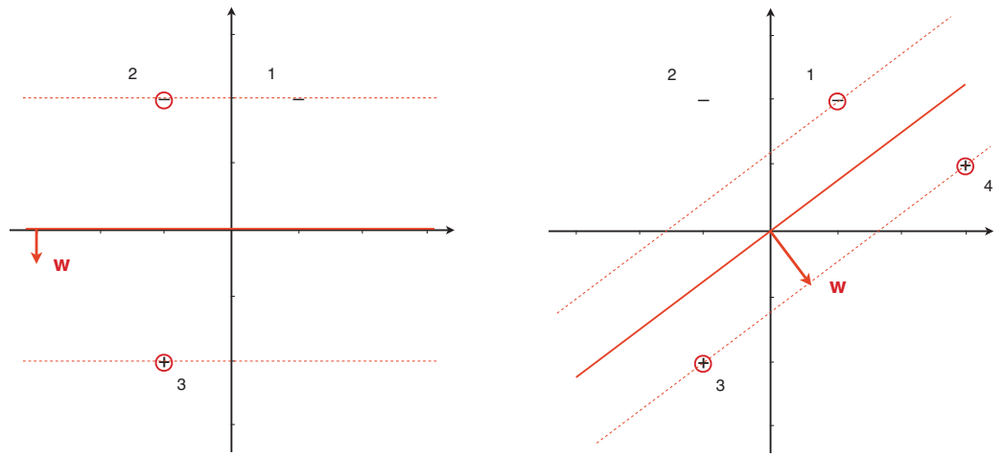



**Figure 7.8.** **(left)** A maximum-margin classifier built from three examples, with $\mathbf{w} = (0, -1/2)$ and margin 2. The circled examples are the support vectors: they receive non-zero Lagrange multipliers and define the decision boundary. **(right)** By adding a second positive the decision boundary is rotated to $\mathbf{w} = (3/5, -4/5)$ and the margin decreases to 1.

---

**Example 7.5 (Two maximum-margin classifiers and their support vectors).** Let the data points and labels be as follows (see Figure 7.8 (left)):

$$\mathbf{X} = \begin{pmatrix} 1 & 2 \\ -1 & 2 \\ -1 & -2 \end{pmatrix} \qquad \mathbf{y} = \begin{pmatrix} -1 \\ -1 \\ +1 \end{pmatrix} \qquad \mathbf{X}' = \begin{pmatrix} -1 & -2 \\ 1 & -2 \\ -1 & -2 \end{pmatrix}$$

The matrix $\mathbf{X}'$ on the right incorporates the class labels; i.e., the rows are $y_i \mathbf{x}_i$. The Gram matrix is (without and with class labels):

$$\mathbf{X}\mathbf{X}^{\mathrm{T}} = \begin{pmatrix} 5 & 3 & -5 \\ 3 & 5 & -3 \\ -5 & -3 & 5 \end{pmatrix} \qquad \mathbf{X}'\mathbf{X}'^{\mathrm{T}} = \begin{pmatrix} 5 & 3 & 5 \\ 3 & 5 & 3 \\ 5 & 3 & 5 \end{pmatrix}$$

The dual optimisation problem is thus

$$\begin{aligned} &\underset{\alpha_1,\alpha_2,\alpha_3}{\arg\max} -\frac{1}{2}\big(5\alpha_1^2 + 3\alpha_1\alpha_2 + 5\alpha_1\alpha_3 + 3\alpha_2\alpha_1 + 5\alpha_2^2 + 3\alpha_2\alpha_3 + 5\alpha_3\alpha_1 \\ &\qquad\qquad + 3\alpha_3\alpha_2 + 5\alpha_3^2\big) + \alpha_1 + \alpha_2 + \alpha_3 \\ &= \underset{\alpha_1,\alpha_2,\alpha_3}{\arg\max} -\frac{1}{2}\big(5\alpha_1^2 + 6\alpha_1\alpha_2 + 10\alpha_1\alpha_3 + 5\alpha_2^2 + 6\alpha_2\alpha_3 + 5\alpha_3^2\big) + \alpha_1 + \alpha_2 + \alpha_3 \end{aligned}$$

subject to $\alpha_1 \geq 0, \alpha_2 \geq 0, \alpha_3 \geq 0$ and $-\alpha_1 - \alpha_2 + \alpha_3 = 0$. While in practice such problems are solved by dedicated quadratic optimisation solvers, here we will show how to solve this toy problem by hand.

Using the equality constraint we can eliminate one of the variables, say $\alpha_3$, and simplify the objective function to

$$\begin{aligned}\underset{\alpha_1,\alpha_2,\alpha_3}{\arg\max} -\frac{1}{2}\left(5\alpha_1^2+6\alpha_1\alpha_2+10\alpha_1(\alpha_1+\alpha_2)+5\alpha_2^2+6\alpha_2(\alpha_1+\alpha_2)+5(\alpha_1+\alpha_2)^2\right)& \\ +2\alpha_1+2\alpha_2& \\ =\underset{\alpha_1,\alpha_2,\alpha_3}{\arg\max} -\frac{1}{2}\left(20\alpha_1^2+32\alpha_1\alpha_2+16\alpha_2^2\right)+2\alpha_1+2\alpha_2&\end{aligned}$$

Setting partial derivatives to 0 we obtain $-20\alpha_1-16\alpha_2+2=0$ and $-16\alpha_1-16\alpha_2+2=0$ (notice that, because the objective function is quadratic, these equations are guaranteed to be linear). We therefore obtain the solution $\alpha_1=0$ and $\alpha_2=\alpha_3=1/8$. We then have $\mathbf{w}=1/8(\mathbf{x}_3-\mathbf{x}_2)=\begin{pmatrix}0\\-1/2\end{pmatrix}$, resulting in a margin of $1/||\mathbf{w}||=2$. Finally, $t$ can be obtained from any support vector, say $\mathbf{x}_2$, since $y_2(\mathbf{w}\cdot\mathbf{x}_2-t)=1$; this gives $-1\cdot(-1-t)=1$, hence $t=0$. The resulting maximum-margin classifier is depicted in Figure 7.8 (left). Notice that the first example $\mathbf{x}_1$ is not a support vector, even though it is on the margin: this is because removing it will not affect the decision boundary.

We now add an additional positive at $(3,1)$. This gives the following data matrices:

$$\mathbf{X}'=\begin{pmatrix}-1&-2\\1&-2\\-1&-2\\3&1\end{pmatrix}\qquad \mathbf{X}'\mathbf{X}'^{\mathrm{T}}=\begin{pmatrix}5&3&5&-5\\3&5&3&1\\5&3&5&-5\\-5&1&-5&10\end{pmatrix}$$

It can be verified by similar calculations to those above that the margin decreases to 1 and the decision boundary rotates to $\mathbf{w}=\begin{pmatrix}3/5\\-4/5\end{pmatrix}$ (Figure 7.8 (right)). The Lagrange multipliers now are $\alpha_1=1/2$, $\alpha_2=0$, $\alpha_3=1/10$ and $\alpha_4=2/5$. Thus, only $\mathbf{x}_3$ is a support vector in both the original and the extended data set.

---

## *Soft margin SVM*

If the data is not linearly separable, then the constraints $\mathbf{w}\cdot\mathbf{x}_i-t\geq 1$ posed by the examples are not jointly satisfiable. However, there is a very elegant way of adapting the optimisation problem such that it admits a solution even in this case. The idea is to introduce *slack variables* $\xi_i$, one for each example, which allow some of them to be inside the margin or even at the wrong side of the decision boundary – we will call these *margin errors*. Thus, we change the constraints to $\mathbf{w}\cdot\mathbf{x}_i-t\geq 1-\xi_i$ and add the sum of

all slack variables to the objective function to be minimised, resulting in the following *soft margin* optimisation problem:

$$\mathbf{w}^*, t^*, \xi_i^* = \underset{\mathbf{w},t,\xi_i}{\arg\min} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i$$
$$\text{subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i - t) \geq 1 - \xi_i \text{ and } \xi_i \geq 0, 1 \leq i \leq n \tag{7.11}$$

$C$ is a user-defined parameter trading off margin maximisation against slack variable minimisation: a high value of $C$ means that margin errors incur a high penalty, while a low value permits more margin errors (possibly including misclassifications) in order to achieve a large margin. If we allow more margin errors we need fewer support vectors, hence $C$ controls to some extent the 'complexity' of the SVM and hence is often referred to as the *complexity parameter*. It can be seen as a form of regularisation similar to that discussed in the context of least-squares regression.

The Lagrange function is then as follows:

$$\begin{aligned}
\Lambda(\mathbf{w}, t, \xi_i, \alpha_i, \beta_i) &= \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n}\alpha_i(y_i(\mathbf{w}\cdot\mathbf{x}_i - t) - (1-\xi_i)) - \sum_{i=1}^{n}\beta_i\xi_i \\
&= \frac{1}{2}\mathbf{w}\cdot\mathbf{w} - \mathbf{w}\cdot\left(\sum_{i=1}^{n}\alpha_i y_i \mathbf{x}_i\right) + t\left(\sum_{i=1}^{n}\alpha_i y_i\right) + \sum_{i=1}^{n}\alpha_i + \sum_{i=1}^{n}(C - \alpha_i - \beta_i)\xi_i \\
&= \Lambda(\mathbf{w}, t, \alpha_i) + \sum_{i=1}^{n}(C - \alpha_i - \beta_i)\xi_i
\end{aligned}$$

For an optimal solution every partial derivative with respect to $\xi_i$ should be 0, from which it follows that $C - \alpha_i - \beta_i = 0$ for all $i$, and hence the added term vanishes from the dual problem. Furthermore, since both $\alpha_i$ and $\beta_i$ are positive, this means that $\alpha_i$ cannot be larger than $C$, which manifests itself as an additional upper bound on $\alpha_i$ in the dual problem:

$$\alpha_1^*, \ldots, \alpha_n^* = \underset{\alpha_1,\ldots,\alpha_n}{\arg\max} -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \mathbf{x}_i\cdot\mathbf{x}_j + \sum_{i=1}^{n}\alpha_i$$
$$\text{subject to } 0 \leq \alpha_i \leq C \text{ and } \sum_{i=1}^{n}\alpha_i y_i = 0 \tag{7.12}$$

This is a remarkable and beautiful result. It follows from the particular way that slack variables were added to the optimisation problem in Equation 7.11. By restricting the slack variables to be positive and adding them to the objective function to be minimised, they function as penalty terms, measuring deviations on the wrong side of the margin only. Furthermore, the fact that the $\beta_i$ multipliers do not appear in the dual objective follows from the fact that the penalty term in the primal objective is linear in $\xi_i$. In effect, these slack variables implement what was called *hinge loss* in Figure 2.6 on p.63: a margin $z > 1$ incurs no penalty, and a margin $z = 1 - \xi \leq 1$ incurs a penalty $\xi = 1 - z$.

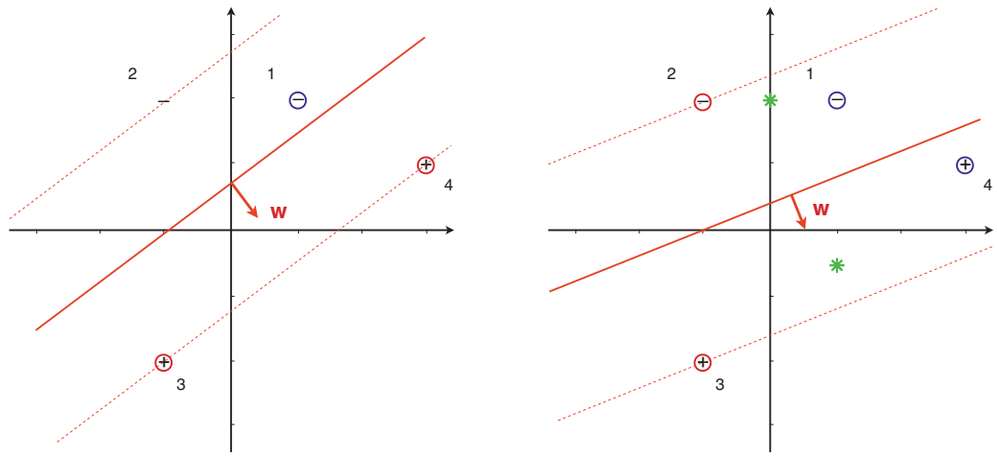



**Figure 7.9. (left)** The soft margin classifier learned with $C = 5/16$, at which point $\mathbf{x}_2$ is about to become a support vector. **(right)** The soft margin classifier learned with $C = 1/10$: all examples contribute equally to the weight vector. The asterisks denote the class means, and the decision boundary is parallel to the one learned by the basic linear classifier.

What is the significance of the upper bound $C$ on the $\alpha_i$ multipliers? Since $C - \alpha_i - \beta_i = 0$ for all $i$, $\alpha_i = C$ implies $\beta_i = 0$. The $\beta_i$ multipliers come from the $\xi_i \geq 0$ constraint, and a multiplier of 0 means that the lower bound is not reached, i.e., $\xi_i > 0$ (analogous to the fact that $\alpha_j = 0$ means that $\mathbf{x}_j$ is not a support vector and hence $\mathbf{w} \cdot \mathbf{x}_j - t > 1$). In other words, a solution to the soft margin optimisation problem in dual form divides the training examples into three cases:

$\alpha_i = 0$ these are outside or on the margin;

$0 < \alpha_i < C$ these are the support vectors on the margin;

$\alpha_i = C$ these are on or inside the margin.

Notice that we still have $\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i$, and so both second and third case examples participate in spanning the decision boundary.

---

**Example 7.6 (Soft margins).** We continue Example 7.5, where we saw that adding the positive example $\mathbf{x}_4 = (3,1)$ to the first three examples significantly reduced the margin from 2 to 1. We will now show that soft margin classifiers with larger margins are learned with sufficiently large complexity parameter $C$.

Recall that the Lagrange multipliers for the classifier in Figure 7.8 (right) are $\alpha_1 = 1/2$, $\alpha_2 = 0$, $\alpha_3 = 1/10$ and $\alpha_4 = 2/5$. So $\alpha_1$ is the largest multiplier, and as long as $C > \alpha_1 = 1/2$ no margin errors are tolerated. For $C = 1/2$ we have $\alpha_1 = C$,

and hence for $C < 1/2$ we have that $\mathbf{x}_1$ becomes a margin error and the optimal classifier is a soft margin classifier. Effectively, with decreasing $C$ the decision boundary and the upper margin move upward, while the lower margin stays the same.

The upper margin reaches $\mathbf{x}_2$ for $C = 5/16$ (Figure 7.9 (left)), at which point we have $\mathbf{w} = \begin{pmatrix} 3/8 \\ -1/2 \end{pmatrix}$, $t = 3/8$ and the margin has increased to 1.6. Furthermore, we have $\xi_1 = 6/8, \alpha_1 = C = 5/16, \alpha_2 = 0, \alpha_3 = 1/16$ and $\alpha_4 = 1/4$.

If we now decrease $C$ further, the decision boundary starts to rotate clockwise, so that $\mathbf{x}_4$ becomes a margin error as well, and only $\mathbf{x}_2$ and $\mathbf{x}_3$ are support vectors. The boundary rotates until $C = 1/10$, at which point we have $\mathbf{w} = \begin{pmatrix} 1/5 \\ -1/2 \end{pmatrix}$, $t = 1/5$ and the margin has increased to 1.86. Furthermore, we have $\xi_1 = 4/10$ and $\xi_4 = 7/10$, and all multipliers have become equal to $C$ (Figure 7.9 (right)).

Finally, when $C$ decreases further the decision boundary stays where it is, but the norm of the weight vector gradually decreases and all points become margin errors.

---

Example 7.6 illustrates an important point: for low enough $C$, all examples receive the same multiplier $C$, and hence we have $\mathbf{w} = C\sum_{i=1}^{n} y_i\mathbf{x}_i = C(Pos \cdot \boldsymbol{\mu}^{\oplus} - Neg \cdot \boldsymbol{\mu}^{\ominus})$, where $\boldsymbol{\mu}^{\oplus}$ and $\boldsymbol{\mu}^{\ominus}$ are the means of the positive and negative examples, respectively. In other words, *a minimal-complexity soft margin classifier summarises the classes by their class means in a way very similar to the basic linear classifier*. For intermediate values of $C$ the decision boundary is spanned by the support vectors and the per-class means of the margin errors.

In summary, support vector machines are linear classifiers that construct the unique decision boundary that maximises the distance to the nearest training examples (the support vectors). The complexity parameter $C$ can be used to adjust the number and severity of allowed margin violations. Training an SVM involves solving a large quadratic optimisation problem and is usually best left to a dedicated numerical solver.

## 7.4 Obtaining probabilities from linear classifiers

As we have seen, a linear classifier produces scores $\hat{s}(\mathbf{x}_i) = \mathbf{w}\cdot\mathbf{x}_i - t$ that are thresholded at 0 in order to classify examples. Owing to the geometric nature of linear classifiers, such scores can be used to obtain the (signed) distance of $\mathbf{x}_i$ from the decision boundary. To see this, notice that the length of the projection of $\mathbf{x}_i$ onto $\mathbf{w}$ is $||\mathbf{x}_i||\cos\theta$, where

# 6

# Kernel Methods

In Chapters 3 and 4, we considered linear parametric models for regression and classification in which the form of the mapping $y(\mathbf{x}, \mathbf{w})$ from input $\mathbf{x}$ to output $y$ is governed by a vector $\mathbf{w}$ of adaptive parameters. During the learning phase, a set of training data is used either to obtain a point estimate of the parameter vector or to determine a posterior distribution over this vector. The training data is then discarded, and predictions for new inputs are based purely on the learned parameter vector $\mathbf{w}$. This approach is also used in nonlinear parametric models such as neural networks.

*Chapter 5*

However, there is a class of pattern recognition techniques, in which the training data points, or a subset of them, are kept and used also during the prediction phase. For instance, the Parzen probability density model comprised a linear combination of 'kernel' functions each one centred on one of the training data points. Similarly, in Section 2.5.2 we introduced a simple technique for classification called nearest neighbours, which involved assigning to each new test vector the same label as the

*Section 2.5.1*

closest example from the training set. These are examples of *memory-based* methods that involve storing the entire training set in order to make predictions for future data points. They typically require a metric to be defined that measures the similarity of any two vectors in input space, and are generally fast to 'train' but slow at making predictions for test data points.

Many linear parametric models can be re-cast into an equivalent 'dual representation' in which the predictions are also based on linear combinations of a *kernel function* evaluated at the training data points. As we shall see, for models which are based on a fixed nonlinear *feature space* mapping $\boldsymbol{\phi}(\mathbf{x})$, the kernel function is given by the relation

$$k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}'). \tag{6.1}$$

From this definition, we see that the kernel is a symmetric function of its arguments so that $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x}', \mathbf{x})$. The kernel concept was introduced into the field of pattern recognition by Aizerman *et al.* (1964) in the context of the method of potential functions, so-called because of an analogy with electrostatics. Although neglected for many years, it was re-introduced into machine learning in the context of large-margin classifiers by Boser *et al.* (1992) giving rise to the technique of *support vector machines*. Since then, there has been considerable interest in this topic, both in terms of theory and applications. One of the most significant developments has been the extension of kernels to handle symbolic objects, thereby greatly expanding the range of problems that can be addressed.

*Chapter 7*

The simplest example of a kernel function is obtained by considering the identity mapping for the feature space in (6.1) so that $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$, in which case $k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^{\mathrm{T}}\mathbf{x}'$. We shall refer to this as the linear kernel.

The concept of a kernel formulated as an inner product in a feature space allows us to build interesting extensions of many well-known algorithms by making use of the *kernel trick*, also known as *kernel substitution*. The general idea is that, if we have an algorithm formulated in such a way that the input vector $\mathbf{x}$ enters only in the form of scalar products, then we can replace that scalar product with some other choice of kernel. For instance, the technique of kernel substitution can be applied to principal component analysis in order to develop a nonlinear variant of PCA (Schölkopf *et al.*, 1998). Other examples of kernel substitution include nearest-neighbour classifiers and the kernel Fisher discriminant (Mika *et al.*, 1999; Roth and Steinhage, 2000; Baudat and Anouar, 2000).

*Section 12.3*

There are numerous forms of kernel functions in common use, and we shall encounter several examples in this chapter. Many have the property of being a function only of the difference between the arguments, so that $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$, which are known as *stationary* kernels because they are invariant to translations in input space. A further specialization involves *homogeneous* kernels, also known as *radial basis functions*, which depend only on the magnitude of the distance (typically Euclidean) between the arguments so that $k(\mathbf{x}, \mathbf{x}') = k(\|\mathbf{x} - \mathbf{x}'\|)$.

*Section 6.3*

For recent textbooks on kernel methods, see Schölkopf and Smola (2002), Herbrich (2002), and Shawe-Taylor and Cristianini (2004).

# 9 Mixture Models and EM

If we define a joint distribution over observed and latent variables, the corresponding distribution of the observed variables alone is obtained by marginalization. This allows relatively complex marginal distributions over observed variables to be expressed in terms of more tractable joint distributions over the expanded space of observed and latent variables. The introduction of latent variables thereby allows complicated distributions to be formed from simpler components. In this chapter, we shall see that mixture distributions, such as the Gaussian mixture discussed in Section 2.3.9, can be interpreted in terms of discrete latent variables. Continuous latent variables will form the subject of Chapter 12.

As well as providing a framework for building more complex probability distributions, mixture models can also be used to cluster data. We therefore begin our discussion of mixture distributions by considering the problem of finding clusters in a set of data points, which we approach first using a nonprobabilistic technique called the $K$-means algorithm (Lloyd, 1982). Then we introduce the latent variable

*Section 9.1*

view of mixture distributions in which the discrete latent variables can be interpreted as defining assignments of data points to specific components of the mixture. A general technique for finding maximum likelihood estimators in latent variable models is the expectation-maximization (EM) algorithm. We first of all use the Gaussian mixture distribution to motivate the EM algorithm in a fairly informal way, and then we give a more careful treatment based on the latent variable viewpoint. We shall see that the $K$-means algorithm corresponds to a particular nonprobabilistic limit of EM applied to mixtures of Gaussians. Finally, we discuss EM in some generality.

*Section 9.2*

*Section 9.3*

*Section 9.4*

Gaussian mixture models are widely used in data mining, pattern recognition, machine learning, and statistical analysis. In many applications, their parameters are determined by maximum likelihood, typically using the EM algorithm. However, as we shall see there are some significant limitations to the maximum likelihood approach, and in Chapter 10 we shall show that an elegant Bayesian treatment can be given using the framework of variational inference. This requires little additional computation compared with EM, and it resolves the principal difficulties of maximum likelihood while also allowing the number of components in the mixture to be inferred automatically from the data.

## 9.1. $K$-means Clustering

We begin by considering the problem of identifying groups, or clusters, of data points in a multidimensional space. Suppose we have a data set $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ consisting of $N$ observations of a random $D$-dimensional Euclidean variable $\mathbf{x}$. Our goal is to partition the data set into some number $K$ of clusters, where we shall suppose for the moment that the value of $K$ is given. Intuitively, we might think of a cluster as comprising a group of data points whose inter-point distances are small compared with the distances to points outside of the cluster. We can formalize this notion by first introducing a set of $D$-dimensional vectors $\boldsymbol{\mu}_k$, where $k = 1, \ldots, K$, in which $\boldsymbol{\mu}_k$ is a prototype associated with the $k^{\text{th}}$ cluster. As we shall see shortly, we can think of the $\boldsymbol{\mu}_k$ as representing the centres of the clusters. Our goal is then to find an assignment of data points to clusters, as well as a set of vectors $\{\boldsymbol{\mu}_k\}$, such that the sum of the squares of the distances of each data point to its closest vector $\boldsymbol{\mu}_k$, is a minimum.

It is convenient at this point to define some notation to describe the assignment of data points to clusters. For each data point $\mathbf{x}_n$, we introduce a corresponding set of binary indicator variables $r_{nk} \in \{0, 1\}$, where $k = 1, \ldots, K$ describing which of the $K$ clusters the data point $\mathbf{x}_n$ is assigned to, so that if data point $\mathbf{x}_n$ is assigned to cluster $k$ then $r_{nk} = 1$, and $r_{nj} = 0$ for $j \neq k$. This is known as the 1-of-$K$ coding scheme. We can then define an objective function, sometimes called a *distortion measure*, given by

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} \|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2 \tag{9.1}$$

which represents the sum of the squares of the distances of each data point to its

assigned vector $\boldsymbol{\mu}_k$. Our goal is to find values for the $\{r_{nk}\}$ and the $\{\boldsymbol{\mu}_k\}$ so as to minimize $J$. We can do this through an iterative procedure in which each iteration involves two successive steps corresponding to successive optimizations with respect to the $r_{nk}$ and the $\boldsymbol{\mu}_k$. First we choose some initial values for the $\boldsymbol{\mu}_k$. Then in the first phase we minimize $J$ with respect to the $r_{nk}$, keeping the $\boldsymbol{\mu}_k$ fixed. In the second phase we minimize $J$ with respect to the $\boldsymbol{\mu}_k$, keeping $r_{nk}$ fixed. This two-stage optimization is then repeated until convergence. We shall see that these two stages of updating $r_{nk}$ and updating $\boldsymbol{\mu}_k$ correspond respectively to the E (expectation) and M (maximization) steps of the EM algorithm, and to emphasize this we shall use the terms E step and M step in the context of the $K$-means algorithm.

*Section 9.4*

Consider first the determination of the $r_{nk}$. Because $J$ in (9.1) is a linear function of $r_{nk}$, this optimization can be performed easily to give a closed form solution. The terms involving different $n$ are independent and so we can optimize for each $n$ separately by choosing $r_{nk}$ to be 1 for whichever value of $k$ gives the minimum value of $\|\mathbf{x}_n - \boldsymbol{\mu}_k\|^2$. In other words, we simply assign the $n^{\text{th}}$ data point to the closest cluster centre. More formally, this can be expressed as

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_j \|\mathbf{x}_n - \boldsymbol{\mu}_j\|^2 \\ 0 & \text{otherwise.} \end{cases} \tag{9.2}$$

Now consider the optimization of the $\boldsymbol{\mu}_k$ with the $r_{nk}$ held fixed. The objective function $J$ is a quadratic function of $\boldsymbol{\mu}_k$, and it can be minimized by setting its derivative with respect to $\boldsymbol{\mu}_k$ to zero giving

$$2\sum_{n=1}^{N} r_{nk}(\mathbf{x}_n - \boldsymbol{\mu}_k) = 0 \tag{9.3}$$

which we can easily solve for $\boldsymbol{\mu}_k$ to give

$$\boldsymbol{\mu}_k = \frac{\sum_n r_{nk}\mathbf{x}_n}{\sum_n r_{nk}}. \tag{9.4}$$

The denominator in this expression is equal to the number of points assigned to cluster $k$, and so this result has a simple interpretation, namely set $\boldsymbol{\mu}_k$ equal to the mean of all of the data points $\mathbf{x}_n$ assigned to cluster $k$. For this reason, the procedure is known as the $K$-*means* algorithm.

The two phases of re-assigning data points to clusters and re-computing the cluster means are repeated in turn until there is no further change in the assignments (or until some maximum number of iterations is exceeded). Because each phase reduces the value of the objective function $J$, convergence of the algorithm is assured. However, it may converge to a local rather than global minimum of $J$. The convergence properties of the $K$-means algorithm were studied by MacQueen (1967).

*Exercise 9.1*

*Appendix A*

The $K$-means algorithm is illustrated using the Old Faithful data set in Figure 9.1. For the purposes of this example, we have made a linear re-scaling of the data, known as *standardizing*, such that each of the variables has zero mean and unit standard deviation. For this example, we have chosen $K = 2$, and so in this

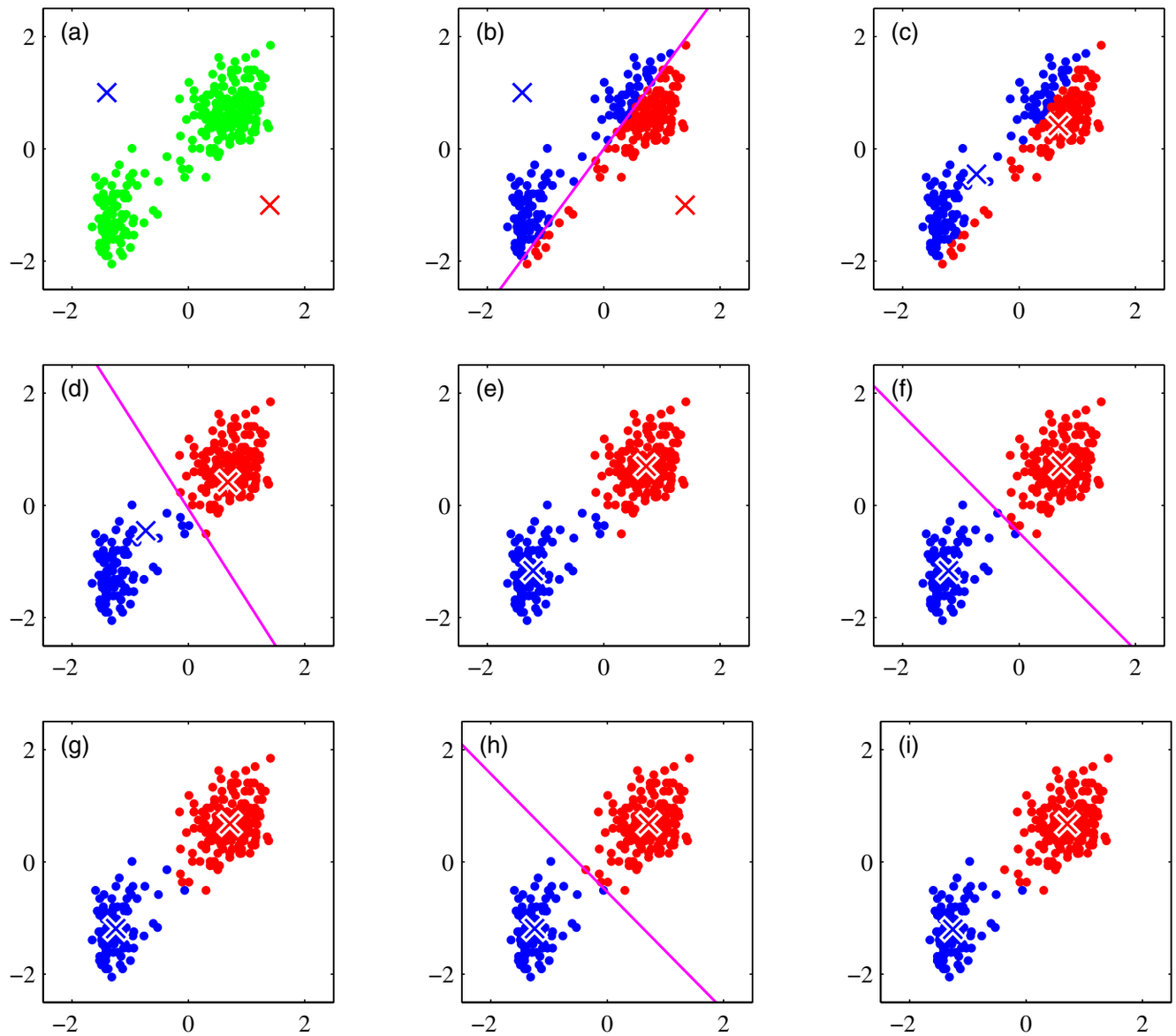



**Figure 9.1** Illustration of the $K$-means algorithm using the re-scaled Old Faithful data set. (a) Green points denote the data set in a two-dimensional Euclidean space. The initial choices for centres $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ are shown by the red and blue crosses, respectively. (b) In the initial E step, each data point is assigned either to the red cluster or to the blue cluster, according to which cluster centre is nearer. This is equivalent to classifying the points according to which side of the perpendicular bisector of the two cluster centres, shown by the magenta line, they lie on. (c) In the subsequent M step, each cluster centre is re-computed to be the mean of the points assigned to the corresponding cluster. (d)–(i) show successive E and M steps through to final convergence of the algorithm.

CHAPTER 8

# Distance-based models

MANY FORMS OF LEARNING are based on generalising from training data to unseen data by exploiting the similarities between the two. With grouping models such as decision trees these similarities take the form of an equivalence relation or partition of the instance space: two instances are similar whenever they end up in the same segment of this partition. In this chapter we consider learning methods that utilise more graded forms of similarity. There are many different ways in which similarity can be measured, and in Section 8.1 we take a look at the most important of them. Section 8.2 is devoted to a discussion of two key concepts in distance-based machine learning: neighbours and exemplars. In Section 8.3 we consider what is perhaps the best-known distance-based learning method: the nearest-neighbour classifier. Section 8.4 investigates $K$-means clustering and close relatives, and Section 8.5 looks at hierarchical clustering by constructing dendrograms. Finally, in Section 8.6 we discuss how several of these methods can be extended using the kind of kernels that we saw in the previous chapter.

## 8.1 So many roads...

It may seem odd at first that there should be many ways to measure distance. I am not referring to the fact that distance can be measured on different scales (kilometres, miles, nautical miles, and so on), as such changes of scale are simple monotonic trans-

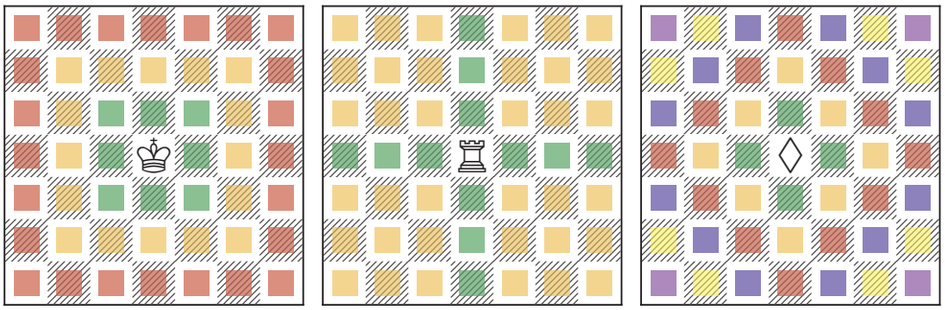


**Figure 8.1.** **(left)** Distance as experienced by a King on a chessboard: green squares are one move away, orange ones two moves and red ones three moves. The shape formed by equidistant squares from the current position is itself a square. **(middle)** A Rook can travel any number of squares in one move, but only horizontally or vertically. No square is further than two moves away. **(right)** The (fictional) KRook combines the restrictions of King and Rook: it can move only one square at a time, and only horizontally or vertically. Equidistant squares now form a lozenge.

formations and do not fundamentally alter the distance measure. A better intuition is obtained by taking the mode of travel into account. Clearly, when travelling from Bristol to Amsterdam by train you travel a larger distance then when travelling by plane, because planes are less restricted in their paths than trains. We will explore this a bit further by considering the game of chess.

In chess, each piece is governed by a set of rules that restrict its possible moves. These restrictions can be directional: for instance, King and Queen can move horizontally, vertically and diagonally, while a Bishop can only move diagonally, a Rook only horizontally and vertically, and pawns only upwards. King and pawn are further restricted by the fact that they can move only one square at a time, whereas Queen, Rook and Bishop can move any number of squares in a single allowed direction. Finally, a Knight moves according to a very specific pattern (one diagonal step and one horizontal or vertical step in a single move).

Although these pieces move around on the same board, they experience distances in very different ways. For example, the next square down is one move away for King, Queen and Rook; three moves away for a Knight; and unreachable for Bishop or pawn. This is, of course, very similar to our experience in the real world. Trains and cars can only move along tracks or roads, like a Bishop, which leaves remote places unreachable. A mountain range can mean large detours when travelling by car, train or on foot, but is easy to cross when flying. On an underground, two stations a few streets away may be only reachable with several changes of line, not unlike the way a Knight can reach a nearby square only in two or three moves. And on foot we are most flexible but

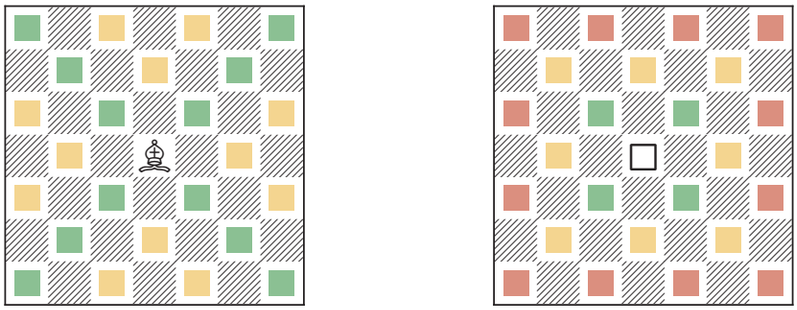


**Figure 8.2. (left)** The Bishop's world: squares are either one or two moves away, or else unreachable. **(right)** The fictional Bing combines the restrictions of King and Bishop: it can only move one square at a time, and only diagonally. Equidistant squares now form a punctuated square.

also slow, like a King.

Figure 8.1 visualises the distances experienced by King and Rook. Both can reach all parts of the chessboard, but a Rook can travel much faster. In fact, a Rook can reach any square in either one or two moves (assuming no other pieces are in its way). All squares one move away form a cross, and the remaining squares are one additional move away. A King will often have to travel more than two moves to reach a particular square (although there are also squares that the King can reach in one move while a Rook needs two). The squares one move away form a small square shape around the current position; those two moves away form a larger square around the smaller square; and so on. Figure 8.1 (right) shows a piece that doesn't exist in chess, but could. It combines the restrictions of King and Rook, and I therefore call it a KRook. Like a King, it can only move one square at a time; and like a Rook, it can only move horizontally and vertically. For the KRook, equidistant squares form a sort of lozenge around the current position.

Figure 8.2 (left) visualises the Bishop's moves. The Bishop is somewhat similar to the Rook in that some squares (those of the same colour as its current square) are never more than two moves away; however, the remaining squares of the other colour are unreachable. Combining the restrictions of the Bishop (only diagonal moves) with those of the King (one square per move) we obtain another fictional piece, the Bing (Figure 8.2 (right)). We could say that the world of Bishops and Bings is rotated 45 degrees, compared with the world of Rooks and KRooks.

What's the relevance of all this when trying to understand distance-based machine learning, you may ask? Well, the rank (row) and file (column) on a chessboard is not unlike a discrete or categorical feature in machine learning (in fact, since ranks and files are ordered, they are ☞*ordinal features*, as we will further discuss in Chapter 10). We can switch to real-valued features by imagining a 'continuous' chessboard with

infinitely many, infinitesimally narrow ranks and files. Squares now become points, and distances are not expressed as the number of squares travelled, but simply as a real number on some scale. If we now look at the shapes obtained by connecting equidistant points, we see that many of these carry over from the discrete to the continuous case. For a King, for example, all points a given fixed distance away still form a square around the current position; and for a KRook they still form a square rotated 45 degrees. As it happens, these are special cases of the following generic concept.

**Definition 8.1 (Minkowski distance).** *If $\mathscr{X} = \mathbb{R}^d$, the* Minkowski distance *of order $p > 0$ is defined as*

$$\mathrm{Dis}_p(\mathbf{x},\mathbf{y}) = \left(\sum_{j=1}^{d} |x_j - y_j|^p\right)^{1/p} = ||\mathbf{x}-\mathbf{y}||_p$$

*where $||\mathbf{z}||_p = \left(\sum_{j=1}^{d} |z_j|^p\right)^{1/p}$ is the $p$-norm (sometimes denoted $L_p$ norm) of the vector $\mathbf{z}$. We will often refer to $\mathrm{Dis}_p$ simply as the $p$-norm.*

So, the 2-*norm* refers to the familiar *Euclidean distance*

$$\mathrm{Dis}_2(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{j=1}^{d} (x_j - y_j)^2} = \sqrt{(\mathbf{x}-\mathbf{y})^{\mathrm{T}}(\mathbf{x}-\mathbf{y})}$$

which measures distance 'as the crow flies'. Two other values of $p$ can be related back to the chess example. The 1-*norm* denotes *Manhattan distance*, also called *cityblock distance*:

$$\mathrm{Dis}_1(\mathbf{x},\mathbf{y}) = \sum_{j=1}^{d} |x_j - y_j|$$

This is the distance if we can only travel along coordinate axes: similar to a taxi in Manhattan or other cities whose streets follow a regular grid pattern, but also the distance experienced by our fictional KRook piece. If we now let $p$ grow larger, the distance will be more and more dominated by the largest coordinate-wise distance, from which we can infer that $\mathrm{Dis}_\infty(\mathbf{x},\mathbf{y}) = \max_j |x_j - y_j|$. This is the distance experienced by the King on a chessboard, who can move diagonally as well as horizontally and vertically but only one step at a time; it is also called *Chebyshev distance*. Figure 8.3 (left) visualises equidistant points from the origin using Minkowski distances of various orders. It can be seen that Euclidean distance is the only Minkowski distance that is rotation-invariant – in other words, special significance is given to the directions of the coordinate axes whenever $p \neq 2$. Minkowski distances do not refer to a particular choice of origin and are therefore translation-invariant, but none of them are scaling-invariant.

You will sometimes see references to the 0-*norm* (or $L_0$ norm) which counts the number of non-zero elements in a vector. The corresponding distance then counts the

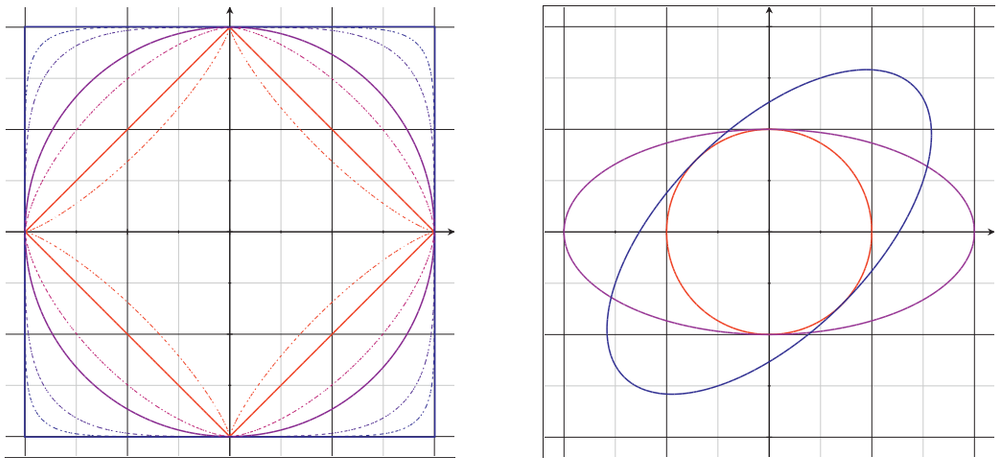


**Figure 8.3. (left)** Lines connecting points at order-$p$ Minkowski distance 1 from the origin for (from inside) $p = 0.8$; $p = 1$ (Manhattan distance, the rotated square in red); $p = 1.5$; $p = 2$ (Euclidean distance, the violet circle); $p = 4$; $p = 8$; and $p = \infty$ (Chebyshev distance, the blue rectangle). Notice that for points on the coordinate axes all distances agree. For the other points, our reach increases with $p$; however, if we require a rotation-invariant distance metric then Euclidean distance is our only choice. **(right)** The rotated ellipse $\mathbf{x}^{\mathrm{T}}\mathbf{R}^{\mathrm{T}}\mathbf{S}^2\mathbf{R}\mathbf{x} = 1/4$; the axis-parallel ellipse $\mathbf{x}^{\mathrm{T}}\mathbf{S}^2\mathbf{x} = 1/4$; and the circle $\mathbf{x}^{\mathrm{T}}\mathbf{x} = 1/4$ ($\mathbf{R}$ and $\mathbf{S}$ as in Example 8.1).

number of positions in which vectors $\mathbf{x}$ and $\mathbf{y}$ differ. This is not strictly a Minkowski distance; however, we can define it as

$$\mathrm{Dis}_0(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{d} (x_j - y_j)^0 = \sum_{j=1}^{d} I[x_j = y_j]$$

under the understanding that $x^0 = 0$ for $x = 0$ and 1 otherwise. This is actually the distance experienced by a Rook on the chessboard: if both rank and file are different the square is two moves away, if only one of them is different the square is one move away. If $\mathbf{x}$ and $\mathbf{y}$ are binary strings, this is also called the *Hamming distance*. Alternatively, we can see the Hamming distance as the number of bits that need to be flipped to change $\mathbf{x}$ into $\mathbf{y}$; for non-binary strings of unequal length this can be generalised to the notion of *edit distance* or *Levenshtein distance*.

Do all of these mathematical constructs make sense as a notion of distance? In order to answer that question we can draw up a list of properties that a proper distance measure should have, such as non-negativity and symmetry. The generally agreed-upon list defines what is known as a metric.

**Definition 8.2 (Distance metric).** *Given an instance space $\mathscr{X}$, a* distance metric *is a function* $\mathrm{Dis} : \mathscr{X} \times \mathscr{X} \to \mathbb{R}$ *such that for any $x, y, z \in \mathscr{X}$:*

1. *distances between a point and itself are zero:* $\mathrm{Dis}(x, x) = 0$;

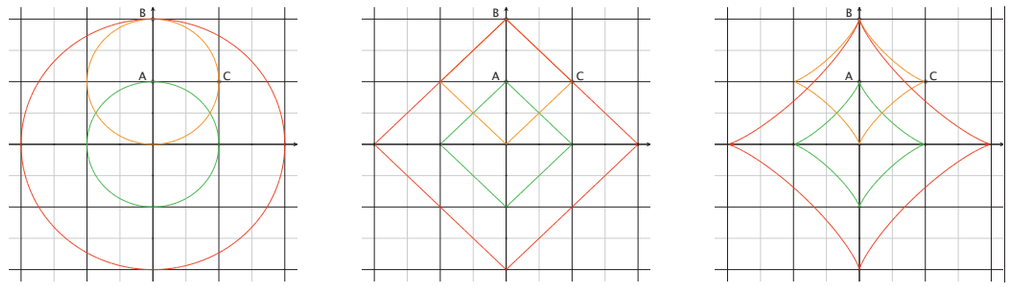



**Figure 8.4. (left)** The green circle connects points the same Euclidean distance (i.e., Minkowski distance of order $p = 2$) away from the origin as A. The orange circle shows that B and C are equidistant from A. The red circle demonstrates that C is closer to the origin than B, which conforms to the triangle inequality. **(middle)** With Manhattan distance ($p = 1$), B and C are equally close to the origin and also equidistant from A. **(right)** With $p < 1$ (here, $p = 0.8$) C is further away from the origin than B; since both are again equidistant from A, it follows that travelling from the origin to C via A is quicker than going there directly, which violates the triangle inequality.

2. *all other distances are larger than zero: if $x \neq y$ then* $\mathrm{Dis}(x, y) > 0$*;*
3. *distances are symmetric:* $\mathrm{Dis}(y, x) = \mathrm{Dis}(x, y)$*;*
4. *detours can not shorten the distance:* $\mathrm{Dis}(x, z) \leq \mathrm{Dis}(x, y) + \mathrm{Dis}(y, z)$.

*If the second condition is weakened to a non-strict inequality – i.e.,* $\mathrm{Dis}(x, y)$ *may be zero even if $x \neq y$ – the function* Dis *is called a* pseudo-metric.

The last condition is called the *triangle inequality* (or sub-additivity, as it really concerns the interaction between distance and addition). Figure 8.4 investigates this for Minkowski distances of various orders. The triangle inequality dictates that the distance from the origin to C is no more than the sum of the distances from the origin to A ($\mathrm{Dis}(\mathrm{O}, \mathrm{A})$) and from A to C ($\mathrm{Dis}(\mathrm{A}, \mathrm{C})$). B is at the same distance from A as C, regardless of the distance measure used; so $\mathrm{Dis}(\mathrm{O}, \mathrm{A}) + \mathrm{Dis}(\mathrm{A}, \mathrm{C})$ is equal to the distance from the origin to B. So, if we draw a circle around the origin through B, the triangle inequality dictates that C not be outside that circle. As we see in the left figure for Euclidean distance, B is the only point where the circles around the origin and around A intersect, so everywhere else the triangle inequality is a strict inequality.

The middle figure shows the same situation for Manhattan distance ($p = 1$). Now, B and C are in fact equidistant from the origin, and so travelling via A to C is no longer a detour, but just one of the many shortest routes. However, if we now decrease $p$ further, we see that C ends up outside the red shape, and is thus further away than B when seen from the origin, whereas of course the sum of the distances from the origin to A and from A to C is still equal to the distance from the origin to B. At this point, our intuition breaks down: Minkowski distances with $p < 1$ are simply not very useful as distances since they all violate the triangle inequality.

Sometimes it is useful to use different scales for different coordinates if they are traversed with different speeds. For instance, for people horizontal distances can be traversed more easily than vertical differences, and consequently it is more realistic to use an ellipse rather than a circle to identify points that can be reached in a fixed amount of time, with the major axis of the ellipse indicating directions that can be traversed at larger speed. The ellipse can also be rotated, so that the major axis is not aligned with any of the coordinates: for instance, this could be the direction of a motorway, or the wind direction. Mathematically, while hyper-spheres (circles in $d \geq 2$ dimensions) of radius $r$ can be defined by the equation $\mathbf{x}^{\mathrm{T}}\mathbf{x} = r^2$, hyper-ellipses are defined by $\mathbf{x}^{\mathrm{T}}\mathbf{M}\mathbf{x} = r^2$, where $\mathbf{M}$ is a matrix describing the appropriate rotation and scaling.

---

**Example 8.1 (Elliptical distance).** Consider the following matrices

$$\mathbf{R} = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix} \qquad \mathbf{S} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1 \end{pmatrix} \qquad \mathbf{M} = \begin{pmatrix} 5/8 & -3/8 \\ -3/8 & 5/8 \end{pmatrix}$$

The matrix $\mathbf{R}$ describes a clockwise rotation of 45 degrees, and the diagonal matrix $\mathbf{S}$ scales the $x$-axis by a factor 1/2. The equation

$$(\mathbf{SRx})^{\mathrm{T}}(\mathbf{SRx}) = \mathbf{x}^{\mathrm{T}}\mathbf{R}^{\mathrm{T}}\mathbf{S}^{\mathrm{T}}\mathbf{SRx} = \mathbf{x}^{\mathrm{T}}\mathbf{R}^{\mathrm{T}}\mathbf{S}^{2}\mathbf{Rx} = \mathbf{x}^{\mathrm{T}}\mathbf{M}\mathbf{x} = 1/4$$

describes a shape which, after clockwise rotation of 45 degrees and scaling of the $x$-axis by a factor 1/2, is a circle with radius 1/2 – i.e., the 'ascending' ellipse in Figure 8.3 (right). The ellipse equation is $(5/8)x^2 + (5/8)y^2 - (3/4)xy = 1/2$.

---

Often, the shape of the ellipse is estimated from data as the inverse of the covariance matrix: $\mathbf{M} = \mathbf{\Sigma}^{-1}$. This leads to the definition of the *Mahalanobis distance*

$$\mathrm{Dis}_M(\mathbf{x}, \mathbf{y}|\mathbf{\Sigma}) = \sqrt{(\mathbf{x}-\mathbf{y})^{\mathrm{T}}\mathbf{\Sigma}^{-1}(\mathbf{x}-\mathbf{y})} \tag{8.1}$$

Using the covariance matrix in this way has the effect of decorrelating and normalising the features, as we saw in Section 7.1. Clearly, Euclidean distance is a special case of Mahalanobis distance with the identity matrix $\mathbf{I}$ as covariance matrix: $\mathrm{Dis}_2(\mathbf{x}, \mathbf{y}) = \mathrm{Dis}_M(\mathbf{x}, \mathbf{y}|\mathbf{I})$.

## 8.2 Neighbours and exemplars

Now that we understand the basics of measuring distance in instance space, we proceed to consider the key ideas underlying distance-based models. The two most

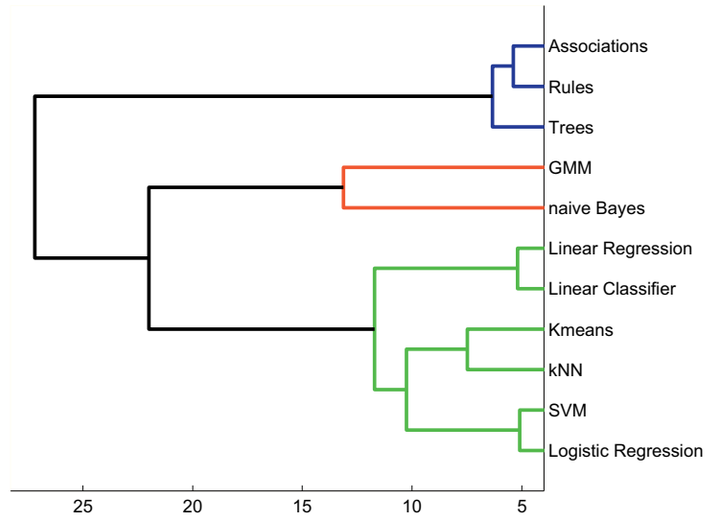



**Figure 8.15.** A dendrogram (printed left to right to improve readability) constructed by hierarchical clustering from the data in Table 1.4 on p.39.

## 8.5 Hierarchical clustering

The clustering methods discussed in the previous section use exemplars to represent a predictive clustering: a partition of the entire instance space. In this section we take a look at methods that represent clusters using trees. We previously encountered ☞*clustering trees* in Section 5.3: those trees use features to navigate the instance space, similar to decision trees, and aren't distance-based as such. Here we consider trees called dendrograms, which are purely defined in terms of a distance measure. Because dendrograms use features only indirectly, as the basis on which the distance measure is calculated, they partition the given data rather than the entire instance space, and hence represent a descriptive clustering rather than a predictive one.

---

**Example 8.6 (Hierarchical clustering of MLM data).** We continue Example 8.4 on p.248. A hierarchical clustering of the MLM data is given in Figure 8.15. The tree shows that the three logical methods at the top form a strong cluster. If we wanted three clusters, we get the logical cluster, a second small cluster {GMM, naive Bayes}, and the remainder. If we wanted four clusters, we would separate GMM and naive Bayes, as the tree indicates this cluster is the least tight of the three (notice that this is slightly different from the solution found by 4-means). If we wanted five clusters, we would construct {Linear Regression, LinearClassifier} as a separate cluster. This illustrates the key

advantage of hierarchical clustering: it doesn't require fixing the number of clusters in advance.

---

A precise definition of a dendrogram is as follows.

**Definition 8.4 (Dendrogram).** *Given a data set $D$, a* dendrogram *is a binary tree with the elements of $D$ at its leaves. An internal node of the tree represents the subset of elements in the leaves of the subtree rooted at that node. The level of a node is the distance between the two clusters represented by the children of the node. Leaves have level 0.*

For this definition to work, we need a way to measure how close two clusters are. You might think that this is straightforward: just calculate the distance between the two cluster means. However, this occasionally leads to problems, as discussed later in this section. Furthermore, taking cluster means as exemplars assumes Euclidean distance, and we may want to use one of the other distance metrics discussed earlier. This has led to the introduction of the so-called linkage function, which is a general way to turn pairwise point distances into pairwise cluster distances.

**Definition 8.5 (Linkage function).** *A* linkage function $L : 2^{\mathscr{X}} \times 2^{\mathscr{X}} \to \mathbb{R}$ *calculates the distance between arbitrary subsets of the instance space, given a distance metric* $\mathrm{Dis} : \mathscr{X} \times \mathscr{X} \to \mathbb{R}$.

The most common linkage functions are as follows:

Single linkage — defines the distance between two clusters as the *smallest* pairwise distance between elements from each cluster.

Complete linkage — defines the distance between two clusters as the *largest* pointwise distance.

Average linkage — defines the cluster distance as the *average* pointwise distance.

Centroid linkage — defines the cluster distance as the point distance between the cluster means.

These linkage functions can be defined mathematically as follows:

$$L_{\text{single}}(A, B) = \min_{x \in A, y \in B} \mathrm{Dis}(x, y)$$

$$L_{\text{complete}}(A, B) = \max_{x \in A, y \in B} \mathrm{Dis}(x, y)$$

$$L_{\text{average}}(A, B) = \frac{\sum_{x \in A, y \in B} \mathrm{Dis}(x, y)}{|A| \cdot |B|}$$

$$L_{\text{centroid}}(A, B) = \mathrm{Dis}\left(\frac{\sum_{x \in A} x}{|A|}, \frac{\sum_{y \in B} y}{|B|}\right)$$

Clearly, all these linkage functions coincide for singleton clusters: $L(\{x\},\{y\}) = \mathrm{Dis}(x,y)$. However, for larger clusters they start to diverge. For example, suppose $\mathrm{Dis}(x,y) < \mathrm{Dis}(x,z)$, then the linkage between $\{x\}$ and $\{y,z\}$ is different in all four cases:

$$\begin{aligned}
L_{\text{single}}(\{x\},\{y,z\}) &= \mathrm{Dis}(x,y)\\
L_{\text{complete}}(\{x\},\{y,z\}) &= \mathrm{Dis}(x,z)\\
L_{\text{average}}(\{x\},\{y,z\}) &= \big(\mathrm{Dis}(x,y)+\mathrm{Dis}(x,z)\big)/2\\
L_{\text{centroid}}(\{x\},\{y,z\}) &= \mathrm{Dis}(x,(y+z)/2)
\end{aligned}$$

The general algorithm to build a dendrogram is given in Algorithm 8.4. The tree is built from the data points upwards and is hence a bottom–up or *agglomerative* algorithm. At each iteration the algorithm constructs a new partition of the data by merging the two nearest clusters together. In general, the HAC algorithm gives different results when different linkage functions are used. Single linkage is the easiest case to understand, as it effectively builds a graph by adding increasingly longer links between points, one at a time, such that ultimately there is a path between any pair of points (hence the term 'linkage'). At any point during this process, the connected components are the clusters found at that iteration, and the linkage of the most recently found cluster is the length of the most recently added link. Hierarchical clustering using single linkage can essentially be done by calculating and sorting all pairwise distances between data points, which requires $O(n^2)$ time for $n$ points. The other linkage functions require at least $O(n^2 \log n)$. Notice that the unoptimised algorithm in Algorithm 8.4 has time complexity $O(n^3)$.

**Algorithm 8.4:** HAC$(D, L)$ – Hierarchical agglomerative clustering.

**Input** : data $D \subseteq \mathscr{X}$; linkage function $L : 2^{\mathscr{X}} \times 2^{\mathscr{X}} \to \mathbb{R}$ defined in terms of distance metric.

**Output** : a dendrogram representing a descriptive clustering of $D$.

```
initialise clusters to singleton data points;
create a leaf at level 0 for every singleton cluster;
repeat
    find the pair of clusters X, Y with lowest linkage l, and merge;
    create a parent of X, Y at level l;
until all data points are in one cluster;
return the constructed binary tree with linkage levels;
```

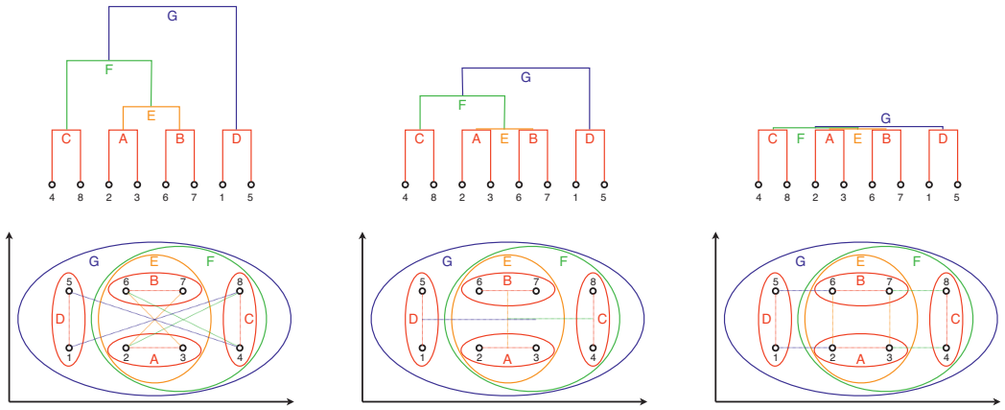



**Figure 8.16.** **(left)** Complete linkage defines cluster distance as the largest pairwise distance between elements from each cluster, indicated by the coloured lines between data points. The clustering found can be represented as nested partitions (bottom) or a dendrogram (top); the level of a horizontal connection between clusters in the dendrogram corresponds to the length of a linkage line. The example assumes that ties are broken by small irregularities in the grid. **(middle)** Centroid linkage defines the distance between clusters as the distance between their means. Notice that E obtains the same linkage as A and B, and so the latter clusters effectively disappear. **(right)** Single linkage defines the distance between clusters as the smallest pairwise distance. The dendrogram all but collapses, which means that no meaningful clusters are found in the given grid configuration.

---

**Example 8.7 (Linkage matters).** We consider a regular grid of 8 points in two rows of four (Figure 8.16). We assume that ties are broken by small irregularities. Each linkage function merges the same clusters in the same order, but the linkages are quite different in each case. Complete linkage gives the impression that D is far removed from the rest, whereas by moving D very slightly to the right it would have been added to E before C. With centroid linkage we see that E has in fact the same linkage as A and B, which means that A and B are not really discernible as separate clusters, even though they are found first. Single linkage seems preferable in this case, as it most clearly demonstrates that there is no meaningful cluster structure in this set of points.

---

Single and complete linkage both define the distance between clusters in terms of a particular pair of points. Consequently, they cannot take the shape of the cluster into account, which is why average and centroid linkage can offer an advantage. However, centroid linkage can lead to non-intuitive dendrograms, as illustrated in Figure

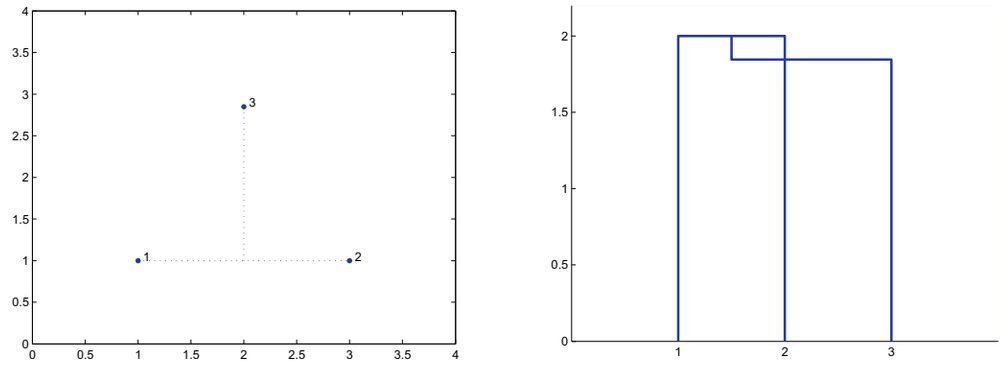



**Figure 8.17. (left)** Points 1 and 2 are closer to each other than to point 3. However, the distance between point 3 to the centroid of the other two points is less than any of the pairwise distances. **(right)** This results in a decrease in linkage when adding point 3 to cluster $\{1,2\}$, and hence a non-monotonic dendrogram.

8.17. The issue here is that we have $L(\{1\},\{2\}) < L(\{1\},\{3\})$ and $L(\{1\},\{2\}) < L(\{2\},\{3\})$ but $L(\{1\},\{2\}) > L(\{1,2\},\{3\})$. The first two inequalities mean that 1 and 2 are the first to be merged into a cluster; but the second inequality means that the level of cluster $\{1,2,3\}$ in the dendrogram drops below the level of $\{1,2\}$. Centroid linkage violates the requirement of *monotonicity*, which stipulates that $L(A,B) < L(A,C)$ and $L(A,B) < L(B,C)$ implies $L(A,B) < L(A \cup B,C)$ for any clusters $A$, $B$ and $C$. The other three linkage functions are monotonic (the example also serves as an illustration why average linkage and centroid linkage are not the same).

Another thing to keep in mind when constructing dendrograms is that the hierarchical clustering method is deterministic and will always construct a clustering. Consider Figure 8.18, which shows a data set of 20 uniformly randomly sampled points. One would be hard-pressed to find any cluster structure in this data; yet a dendrogram constructed with complete linkage and Euclidean distance appears to indicate that there are three or four clearly discernible clusters. But if we look closer, we see that the linkage levels are very close together in the bottom of the tree, and the fact that linkages are higher towards the top comes primarily from the use of complete linkage, which concentrates on maximal pairwise distances. The silhouette in Figure 8.18 (right) confirms that the cluster structure is not very strong. Effectively, we are witnessing here a particular, clustering-related kind of overfitting, already familiar from other tree-based models discussed in Chapter 5. Furthermore, dendrograms – like other tree models – have high variance in that small changes in the data points can lead to large changes in the dendrogram.

In conclusion, hierarchical clustering methods have the distinct advantage that the number of clusters does not need to be fixed in advance. However, this advantage comes at considerable computational cost. Furthermore, we now need to choose not

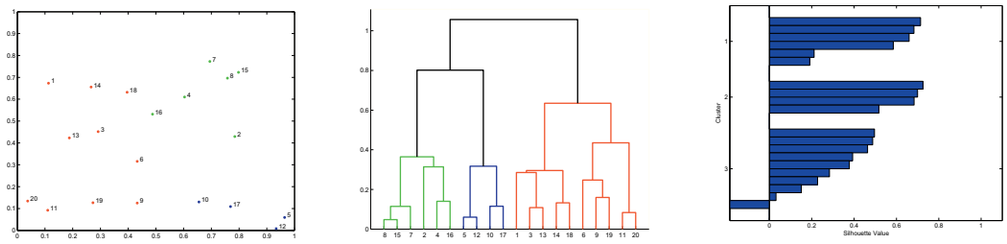



**Figure 8.18. (left)** 20 data points, generated by uniform random sampling. **(middle)** The dendrogram generated from complete linkage. The three clusters suggested by the dendrogram are spurious as they cannot be observed in the data. **(right)** The rapidly decreasing silhouette values in each cluster confirm the absence of a strong cluster structure. Point 18 has a negative silhouette value as it is on average closer to the green points than to the other red points.

just the distance measure used, but also the linkage function.

## 8.6 From kernels to distances

In Section 7.5 we discussed how kernels can be used to extend the power of linear models considerably. Recall that a kernel is a function $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$ that calculates a dot product in some feature space, but without constructing the feature vectors $\phi(\mathbf{x})$ explicitly. Any learning method that can be defined purely in terms of dot products of data points is amenable to such 'kernelisation'. Because of the close connection between Euclidean distance and dot products we can apply the same 'kernel trick' to many distance-based learning methods.

The key insight is that Euclidean distance can be rewritten in terms of dot products:

$$\mathrm{Dis}_2(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}||_2 = \sqrt{(\mathbf{x} - \mathbf{y}) \cdot (\mathbf{x} - \mathbf{y})} = \sqrt{\mathbf{x} \cdot \mathbf{x} - 2\mathbf{x} \cdot \mathbf{y} + \mathbf{y} \cdot \mathbf{y}}$$

This formula clearly shows that the distance between $\mathbf{x}$ and $\mathbf{y}$ decreases whenever the dot product $\mathbf{x} \cdot \mathbf{y}$ increases, which suggests that the dot product itself is a kind of similarity measure. However, it is not translation-invariant, because it depends on the location of the origin. The two terms $\mathbf{x} \cdot \mathbf{x}$ and $\mathbf{y} \cdot \mathbf{y}$ have the effect of making the overall expression translation-invariant. Replacing the dot product with a kernel function $\kappa$, we can construct the following kernelised distance:

$$\mathrm{Dis}_\kappa(\mathbf{x}, \mathbf{y}) = \sqrt{\kappa(\mathbf{x}, \mathbf{x}) - 2\kappa(\mathbf{x}, \mathbf{y}) + \kappa(\mathbf{y}, \mathbf{y})} \tag{8.5}$$

It turns out that $\mathrm{Dis}_\kappa$ defines a pseudo-metric (see Definition 8.2 on p.235) whenever $\kappa$ is a positive semi-definite kernel.[2]

[2] It is only a metric if the feature mapping $\phi$ is injective: suppose not, then some distinct $\mathbf{x}$ and $\mathbf{y}$ are mapped to the same feature vector $\phi(\mathbf{x}) = \phi(\mathbf{y})$, from which we derive $\kappa(\mathbf{x}, \mathbf{x}) - 2\kappa(\mathbf{x}, \mathbf{y}) + \kappa(\mathbf{y}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}) - 2\phi(\mathbf{x}) \cdot \phi(\mathbf{y}) + \phi(\mathbf{y}) \cdot \phi(\mathbf{y}) = 0$.

# Performance Evaluation Metrics

A detailed explanation of model evaluation metrics to evaluate a classification machine learning model.

Predictive models have become a trusted advisor to many businesses and for a good reason. These models can “foresee the future”, and there are many different methods available, meaning any industry can find one that fits their particular challenges.

When we talk about predictive models, we are talking either about a **regression model** (continuous output) or a **classification model** (nominal or binary output). In classification problems, we use two types of algorithms (dependent on the kind of output it creates):

1. **Class output**: Algorithms like SVM and KNN create a class output. For instance, in a binary classification problem, the outputs will be either 0 or 1. However, today we have algorithms that can convert these class outputs to probability.
2. **Probability output**: Algorithms like Logistic Regression, Random Forest, Gradient Boosting, Adaboost, etc. give probability outputs. Converting probability outputs to class output is just a matter of creating a threshold probability.

## Introduction

While data preparation and training a machine learning model is a key step in the machine learning pipeline, it’s equally important to measure the performance of this trained model. How well the model generalizes on the unseen data is what defines adaptive vs non-adaptive machine learning models.

By using different metrics for performance evaluation, we should be in a position to improve the overall predictive power of our model before we roll it out for production on unseen data.

Without doing a proper evaluation of the ML model using different metrics, and depending only on accuracy, it can lead to a problem when the respective model is deployed on unseen data and can result in poor predictions.

This happens because, in cases like these, our models don’t learn but instead memorize; hence, they cannot generalize well on unseen data.

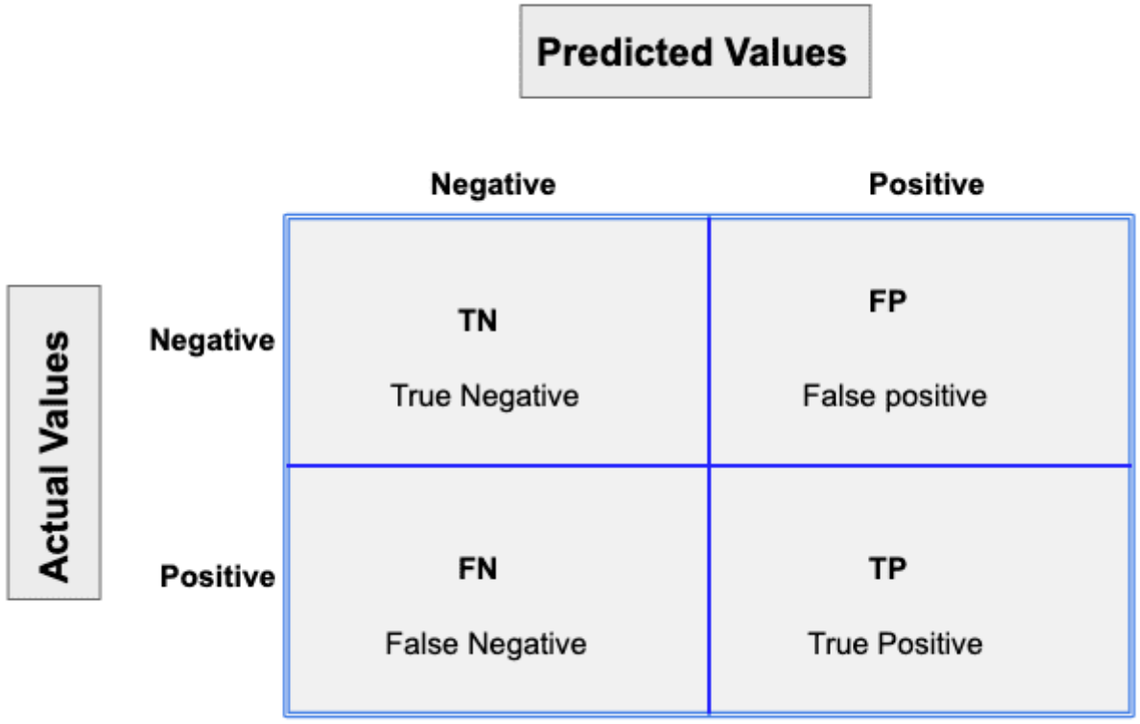
## Model Evaluation Metrics

Let us now define the evaluation metrics for evaluating the performance of a machine learning model, which is an integral component of any data science project. It aims to estimate the generalization accuracy of a model on the future (unseen/out-of-sample) data.

## Confusion Matrix

A confusion matrix is a matrix representation of the prediction results of any binary testing that is often used to **describe the performance of the classification model** (or "classifier") on a set of test data for which the true values are known.

The confusion matrix itself is relatively simple to understand, but the related terminology can be confusing.




Confusion matrix with 2 class labels.

Each prediction can be one of the four outcomes, based on how it matches up to the actual value:

- **True Positive (TP):** Predicted True and True in reality.
- **True Negative (TN):** Predicted False and False in reality.
- **False Positive (FP):** Predicted True and False in reality.
- **False Negative (FN):** Predicted False and True in reality.

Now let us understand this concept using hypothesis testing.

A **Hypothesis** is speculation or theory based on insufficient evidence that lends itself to further testing and experimentation. With further testing, a hypothesis can usually be proven true or false.

A **Null Hypothesis** is a hypothesis that says there is no statistical significance between the two variables in the hypothesis. It is the hypothesis that the researcher is trying to disprove.

> *We would always reject the null hypothesis when it is false, and we would accept the null hypothesis when it is indeed true.*

Even though hypothesis tests are meant to be reliable, **there are two types of errors that can occur.**

These errors are known as **Type 1 and Type II errors.**

For example, when examining the effectiveness of a drug, the null hypothesis would be that the drug does not affect a disease.

**Type I Error:-** equivalent to False Positives(FP).

The first kind of error that is possible involves the rejection of a null hypothesis that is true.

Let's go back to the example of a drug being used to treat a disease. If we reject the null hypothesis in this situation, then we claim that the drug does have some effect on a disease. But if the null hypothesis is true, then, in reality, the drug does not combat the disease at all. The drug is falsely claimed to have a positive effect on a disease.

**Type II Error:-** equivalent to False Negatives(FN).

The other kind of error that occurs when we accept a false null hypothesis. This sort of error is called a type II error and is also referred to as an error of the second kind.

If we think back again to the scenario in which we are testing a drug, what would a type II error look like? A type II error would occur if we accepted that the drug hs no effect on disease, but in reality, it did.

A sample python implementation of the Confusion matrix.

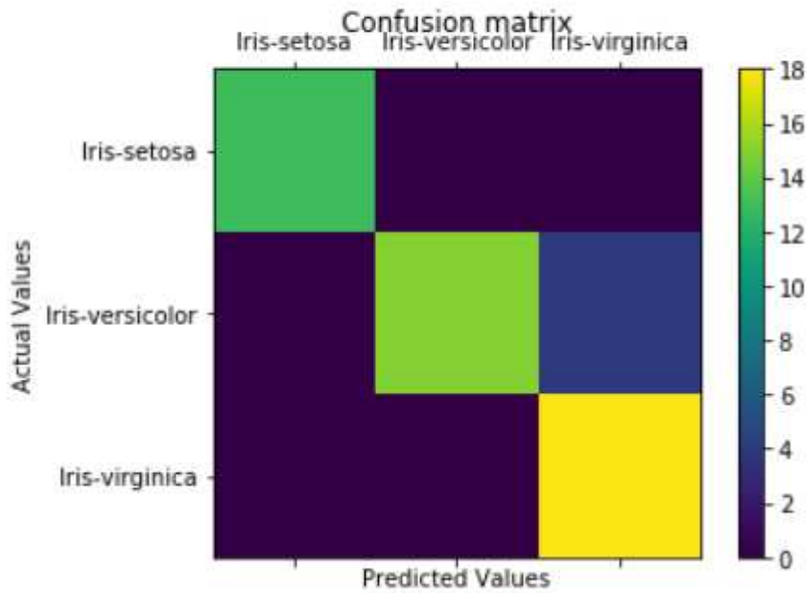
```
import warnings
import pandas as pd
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
%matplotlib inline

#ignore warnings
warnings.filterwarnings('ignore')
# Load digits dataset
url = "http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
df = pd.read_csv(url)
# df = df.values
```

```
X = df.iloc[:,0:4]
y = df.iloc[:,4]
#test size
test_size = 0.33
#generate the same set of random numbers
seed = 7
#Split data into train and test set.
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=test_size,
random_state=seed)
#Train Model
model = LogisticRegression()
model.fit(X_train, y_train)
pred = model.predict(X_test)

#Construct the Confusion Matrix
labels = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
cm = confusion_matrix(y_test, pred, labels)
print(cm)
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(cm)
plt.title('Confusion matrix')
fig.colorbar(cax)
ax.set_xticklabels(['] + labels)
ax.set_yticklabels(['] + labels)
```

```
plt.xlabel('Predicted Values')
plt.ylabel('Actual Values')
plt.show()
```

view raw confusion+matrix.py hosted with ❤ by GitHub

```
[[13  0  0]
 [ 0 15  4]
 [ 0  0 18]]
```




Confusion matrix with 3 class labels.

The diagonal elements represent the number of points for which the predicted label is equal to the true label, while anything off the diagonal was mislabeled by the classifier. Therefore, the higher the diagonal values of the confusion matrix the better, indicating many correct predictions.

In our case, the classifier predicted all the 13 setosa and 18 virginica plants in the test data perfectly. However, it incorrectly classified 4 of the versicolor plants as virginica.

There is also a list of rates that are often computed from a confusion matrix for a binary classifier:

## 1. Accuracy

Overall, how often is the classifier correct?

*Accuracy = (TP+TN)/total*

When our classes are roughly equal in size, we can use accuracy**,** which will give us correctly classified values.

Accuracy is a common evaluation metric for classification problems. It's the number of correct predictions made as a ratio of all predictions made.

**Misclassification Rate(Error Rate):** Overall, how often is it wrong. Since accuracy is the percent we correctly classified (success rate), it follows that our error rate (the percentage we got wrong) can be calculated as follows:

*Misclassification Rate = (FP+FN)/total*

We use the sklearn module to compute the accuracy of a classification task, as shown below.

```
#import modules
import warnings
import pandas as pd
import numpy as np
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.metrics import accuracy_score
#ignore warnings
warnings.filterwarnings('ignore')
# Load digits dataset
iris = datasets.load_iris()
# # Create feature matrix
X = iris.data
# Create target vector
y = iris.target
#test size
test_size = 0.33
```

```
#generate the same set of random numbers
seed = 7
#cross-validation settings
kfold = model_selection.KFold(n_splits=10, random_state=seed)
#Model instance
model = LogisticRegression()
#Evaluate model performance
scoring = 'accuracy'
results = model_selection.cross_val_score(model, X, y, cv=kfold, scoring=scoring)
print('Accuracy -val set: %.2f%% (%.2f)' % (results.mean()*100, results.std()))

#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=test_size,
random_state=seed)
#fit model
model.fit(X_train, y_train)
#accuracy on test set
result = model.score(X_test, y_test)
print("Accuracy - test set: %.2f%%" % (result*100.0))
```

view raw classification_Accuracy.py hosted with ♥ by GitHub

The classification accuracy is **88%** on the validation set.

## 2. Precision

When it predicts yes, how often is it correct?

*Precision=TP/predicted yes*

When we have a class imbalance, accuracy can become an unreliable metric for measuring our performance. For instance, if we had a 99/1 split between two classes, A and B, where the rare event, B, is our positive class, we could build a model that was 99% accurate by just saying everything belonged to class A. Clearly, we shouldn't bother building a model if it doesn't do anything to identify class B; thus, we need different metrics that will discourage this behavior. For this, we use precision and recall instead of accuracy.

## 3. Recall or Sensitivity

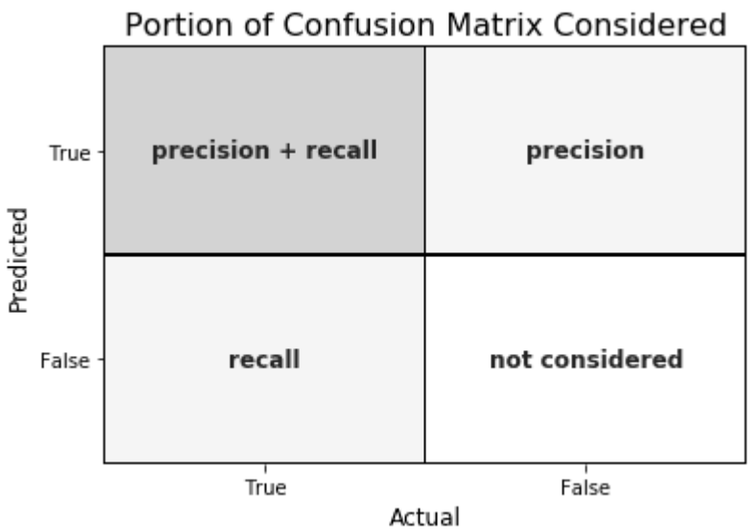
When it's actually yes, how often does it predict yes?

*True Positive Rate = TP/actual yes*

Recall gives us the **true positive rate** (**TPR**), which is the ratio of true positives to everything positive.

In the case of the 99/1 split between classes A and B, the model that classifies everything as A would have a recall of 0% for the positive class, B (precision would be undefined — 0/0). Precision and recall provide a better way of evaluating model performance in the face of a class imbalance. They will correctly tell us that the model has little value for our use case.

Just like accuracy, both precision and recall are easy to compute and understand but require thresholds. Besides, precision and recall only consider half of the confusion matrix:




## 4. F1 Score

The F1 score is the harmonic mean of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

***Why harmonic mean?*** Since the harmonic mean of a list of numbers skews strongly toward the least elements of the list, it tends (compared to the arithmetic mean) to mitigate the impact of large outliers and aggravate the impact of small ones.

An F1 score punishes extreme values more. Ideally, an F1 Score could be an effective evaluation metric in the following classification scenarios:

- *When FP and FN are equally costly — meaning they miss on true positives or find false positives — both impact the model almost the same way, as in our cancer detection classification example*
- *Adding more data doesn't effectively change the outcome effectively*
- *TN is high (like with flood predictions, cancer predictions, etc.)*

A sample python implementation of the F1 score.

```
import warnings
import pandas
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
from sklearn.metrics import precision_recall_fscore_support as score, precision_score,
recall_score, f1_score

warnings.filterwarnings('ignore')

url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-
diabetes.data.csv"
dataframe = pandas.read_csv(url)
dat = dataframe.values
X = dat[:,:-1]
y = dat[:,-1]
test_size = 0.33
```

```
seed = 7

model = LogisticRegression()
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=test_size, random_state=seed)
model.fit(X_train, y_train)
precision = precision_score(y_test, pred)
print('Precision: %f' % precision)
# recall: tp / (tp + fn)
recall = recall_score(y_test, pred)
print('Recall: %f' % recall)
# f1: tp / (tp + fp + fn)
f1 = f1_score(y_test, pred)
print('F1 score: %f' % f1)
```

view raw F1-Score.py hosted with ❤ by GitHub

```
Precision: 0.696970
Recall: 0.541176
F1 score: 0.609272
```

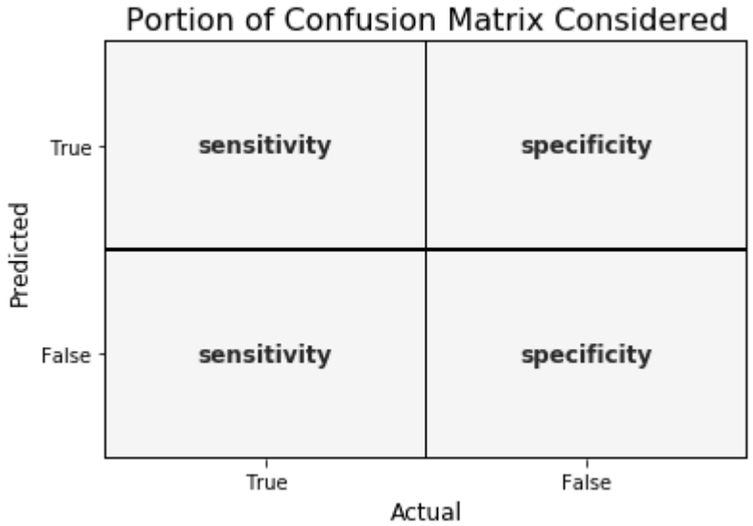
## 5. Specificity

When it's no, how often does it predict no?

*True Negative Rate=TN/actual no*

It is the **true negative rate** or the proportion of true negatives to everything that should have been classified as negative.

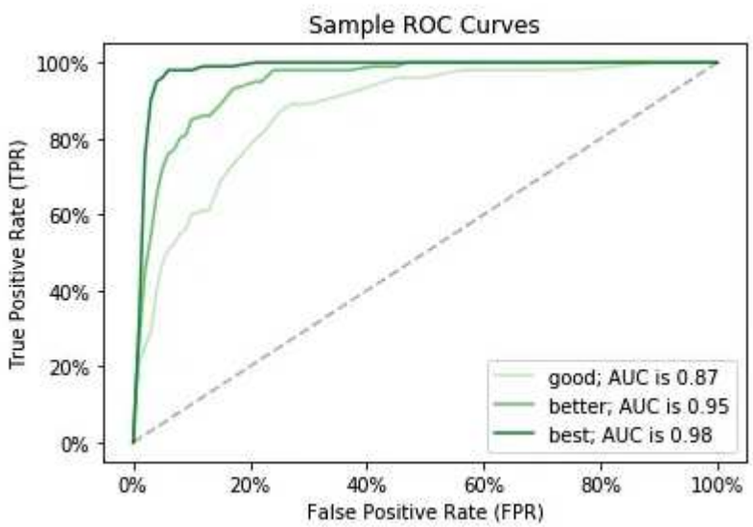
Note that, together, specificity and sensitivity consider the full confusion matrix:




## 6. Receiver Operating Characteristics (ROC) Curve

Measuring the area under the ROC curve is also a very useful method for evaluating a model. By plotting the true positive rate (sensitivity) versus the false-positive rate (1 — specificity), we get the **Receiver Operating Characteristic** (**ROC**) **curve**. This curve allows us to visualize the trade-off between the true positive rate and the false positive rate.

The following are examples of good ROC curves. The dashed line would be random guessing (no predictive value) and is used as a baseline; anything below that is considered worse than guessing. We want to be toward the top-left corner:




A sample python implementation of the ROC curves.

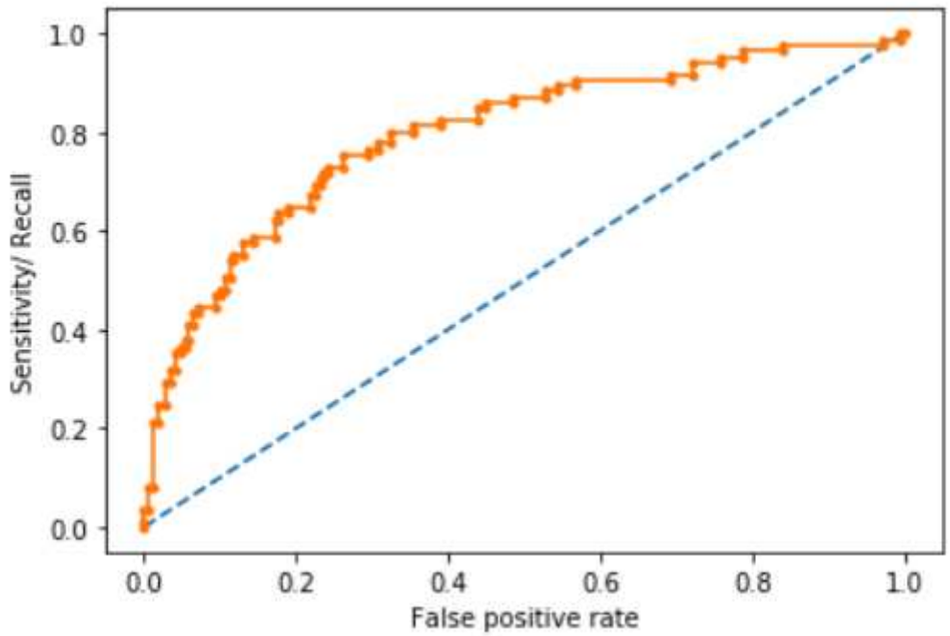
```
#Classification Area under curve

import warnings
```

```
import pandas
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, roc_curve

warnings.filterwarnings('ignore')

url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"
dataframe = pandas.read_csv(url)
dat = dataframe.values
X = dat[:,:-1]
y = dat[:,-1]
seed = 7
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=test_size, random_state=seed)
model.fit(X_train, y_train)

# predict probabilities
probs = model.predict_proba(X_test)
# keep probabilities for the positive outcome only
probs = probs[:, 1]

auc = roc_auc_score(y_test, probs)
print('AUC - Test Set: %.2f%%' % (auc*100))

# calculate roc curve
```

```
fpr, tpr, thresholds = roc_curve(y_test, probs)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
plt.xlabel('False positive rate')
plt.ylabel('Sensitivity/ Recall')
# show the plot
plt.show()
```

view raw AUC-ROC.py hosted with ❤ by GitHub

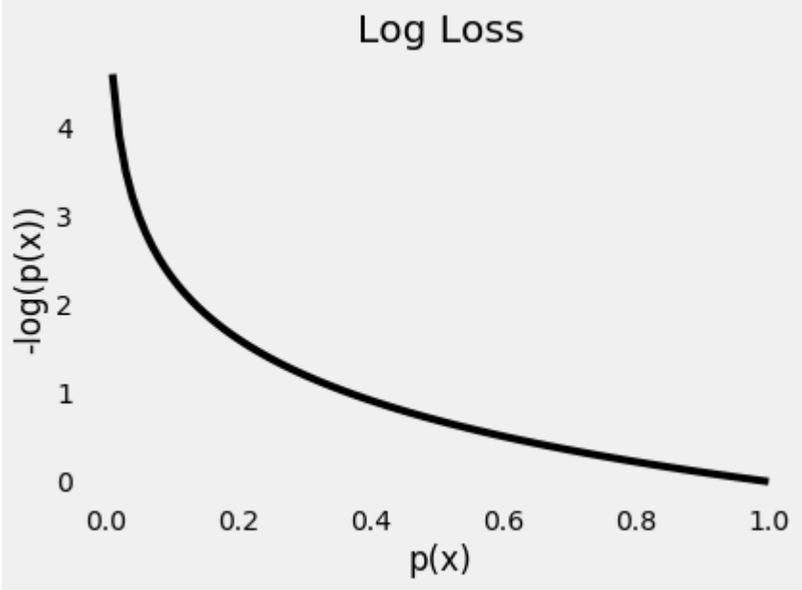
```
AUC - Test Set: 79.65%
```




In the example above, the AUC is relatively close to 1 and greater than 0.5. A perfect classifier will have the ROC curve go along the Y-axis and then along the X-axis.

## Log Loss

Log Loss is the most important classification metric based on probabilities.




As the **predicted probability** of the **true class** gets **closer to zero**, the **loss increases exponentially**

It measures the performance of a classification model where the prediction input is a probability value between 0 and 1. Log loss increases as the predicted probability diverge from the actual label. The goal of any machine learning model is to minimize this value. As such, smaller log loss is better, with a perfect model having a log loss of 0.

A sample python implementation of the Log Loss.

```
#Classification LogLoss
import warnings
import pandas
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss

warnings.filterwarnings('ignore')
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"
dataframe = pandas.read_csv(url)
dat = dataframe.values
X = dat[:,:-1]
```

```
y = dat[:,-1]
seed = 7
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=test_size, random_state=seed)
model.fit(X_train, y_train)
#predict and compute logloss
pred = model.predict(X_test)
accuracy = log_loss(y_test, pred)
print("Logloss: %.2f" % (accuracy))
```

view raw logg_loss.py hosted with ❤ by GitHub

```
Logloss: 8.02
```

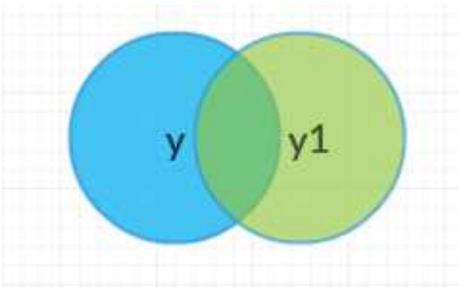
## Jaccard Index

Jaccard Index is one of the simplest ways to calculate and find out the accuracy of a classification ML model. Let's understand it with an example. Suppose we have a labeled test set, with labels as –

```
y = [0,0,0,0,0,1,1,1,1,1]
```

And our model has predicted the labels as –

```
y1 = [1,1,0,0,0,1,1,1,1,1]
```




The above Venn diagram shows us the labels of the test set and the labels of the predictions, and their intersection and union.

Jaccard Index or Jaccard similarity coefficient is a statistic used in understanding the similarities between sample sets. The measurement emphasizes the similarity between finite sample sets and is formally defined as the size of the intersection divided by the size of the union of the two labeled sets, with formula as –

$$J(y, y1) = \frac{|y \cap y1|}{|y \cup y1|} = \frac{|y \cap y1|}{|y| + |y1| - |y \cap y1|}$$

So, for our example, we can see that the intersection of the two sets is equal to 8 (since eight values are predicted correctly) and the union is **10 + 10–8 = 12**. So, the Jaccard index gives us the accuracy as –

$$J(y, y1) = \frac{8}{12} = 0.66$$

So, the accuracy of our model, according to **Jaccard Index**, becomes 0.66, or 66%.

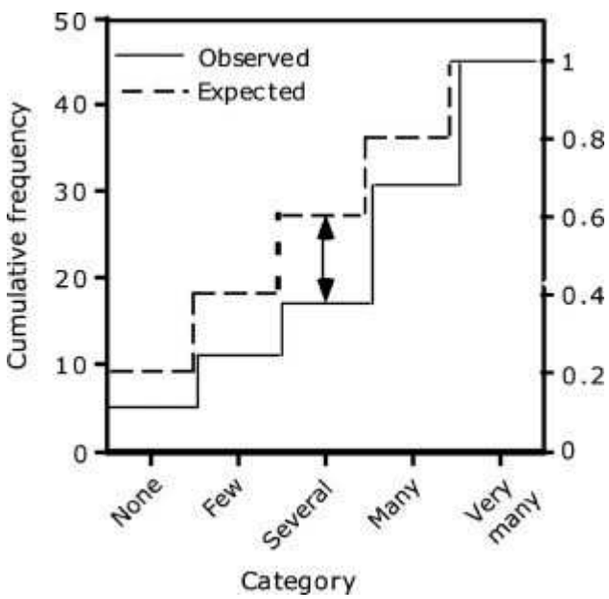
Higher the Jaccard index higher the accuracy of the classifier.

A sample python implementation of the Jaccard index.

```
Jaccard Similarity Score : 0.375
```

## Kolomogorov Smirnov chart

K-S or Kolmogorov-Smirnov chart measures the performance of classification models. More accurately, K-S is a measure of the degree of separation between positive and negative distributions.



The `K-S is 100` if the scores partition the population into two separate groups in which one group contains all the positives and the other all the negatives. On the other hand, If the model cannot differentiate between positives and negatives, then it is as if the model selects cases randomly from the population. The `K-S would be 0`.

In most classification models the K-S will fall between 0 and 100, and that the higher the value the better the model is at separating the positive from negative cases.

The K-S may also be used to test whether two underlying one-dimensional probability distributions differ. It is a very efficient way to determine if two samples are significantly different from each other.

A sample python implementation of the Kolmogorov-Smirnov.

```
Enter outcomes:
[0.44327990912764315, 0.6605825795691285, 0.26203783048580187, 0.306659267902726, 0.2977782138392654, 0.5574513908838
086, 0.9644789110570309, 0.11655821363290042, 0.40892065420391954, 0.9444452718151184]
KstestResult(statistic=0.5463949236854183, pvalue=0.002504270027532512)
```

The Null hypothesis used here assumes that the numbers follow the normal distribution. It returns statistics and p-value. If the **p-value is < alpha**, we reject the Null hypothesis.

*Alpha is defined as the probability of rejecting the null hypothesis given the null hypothesis(H0) is true. For most of the practical applications, alpha is chosen as 0.05.*

## Gain and Lift Chart

Gain or Lift is a measure of the effectiveness of a classification model calculated as the ratio between the results obtained with and without the model. Gain and lift charts are visual aids for evaluating the performance of classification models. However, in contrast to the confusion matrix that evaluates models on the whole population gain or lift chart evaluates model performance in a portion of the population.

The higher the lift (i.e. the further up it is from the baseline), the better the model.

The following gains chart, run on a validation set, shows that with 50% of the data, the model contains 90% of targets, Adding more data adds a negligible increase in the percentage of targets included in the model.

Lift charts are often shown as a **cumulative lift chart**, which is also known as a **gains chart**. Therefore, gains charts are sometimes (perhaps confusingly) called “lift charts”, but they are more accurately *cumulative* lift charts.

It is one of their most common uses is in marketing, to decide if a prospective client is worth calling.

## Gini Coefficient

The Gini coefficient or Gini Index is a popular metric for imbalanced class values. The coefficient

ranges from 0 to 1 where 0 represents perfect equality and 1 represents perfect inequality. Here, if the value of an index is higher, then the data will be more dispersed.

Gini coefficient can be computed from the area under the ROC curve using the following formula:

*Gini Coefficient = (2 * ROC_curve) — 1*

## Conclusion

Understanding how well a machine learning model is going to perform on unseen data is the ultimate purpose behind working with these evaluation metrics. Metrics like accuracy, precision, recall are good ways to evaluate classification models for balanced datasets, but if the data is imbalanced and there's a class disparity, then other methods like ROC/AUC, Gini coefficient perform better in evaluating the model performance.

# Validation Methods

## Introduction

Whenever we build any machine learning model, we feed it with initial data to train the model. And then we feed some unknown data (test data) to understand how well the model performs and generalized over unseen data. If the model performs well on the unseen data, it's consistent and is able to predict with good accuracy on a wide range of input data; then this model is stable.

But this is not the case always! Machine learning models are not always stable and we have to evaluate the stability of the machine learning model. That is where Cross Validation comes into the picture.

> *"In simple terms, Cross-Validation is a technique used to assess how well our Machine learning models perform on unseen data"*

According to Wikipedia, Cross-Validation is the process of assessing how the results of a statistical analysis will generalize to an independent data set.

There are many ways to perform Cross-Validation and we will learn about 4 methods in this article.

Let's first understand the need for Cross-Validation!

## Why do we need Cross-Validation?

Suppose you build a machine learning model to solve a problem, and you have trained the model on a given dataset. When you check the accuracy of the model on the training data, it is close to 95%. Does this mean that your model has trained very well, and it is the best model because of the high accuracy?

No, it's not! Because your model is trained on the given data, it knows the data well, captured even the minute variations(noise), and has generalized very well over the given data. If you expose the model to completely new, unseen data, it might not predict with the same accuracy and it might fail to generalize over the new data. This problem is called over-fitting.

Sometimes the model doesn't train well on the training set as it's not able to find patterns. In this case, it wouldn't perform well on the test set as well. This problem is called Under-fitting.

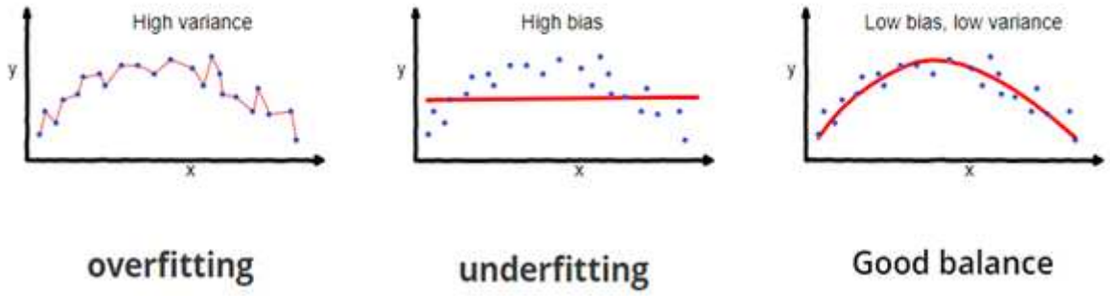



Image Source: fireblazeaischool.in

To overcome over-fitting problems, we use a technique called Cross-Validation.

# What is Cross Validation?

**Cross-Validation** is a resampling technique with the fundamental idea of splitting the dataset into 2 parts- training data and test data. Train data is used to train the model and the unseen test data is used for prediction. If the model performs well over the test data and gives good accuracy, it means the model hasn't overfitted the training data and can be used for prediction.

Let's dive deep and learn about some of the model evaluation techniques.

# 1. Hold Out method

This is the simplest evaluation method and is widely used in Machine Learning projects. Here the entire dataset(population) is divided into 2 sets – train set and test set. The data can be divided into 70-30 or 60-40, 75-25 or 80-20, or even 50-50 depending on the use case. As a rule, the proportion of training data has to be larger than the test data.

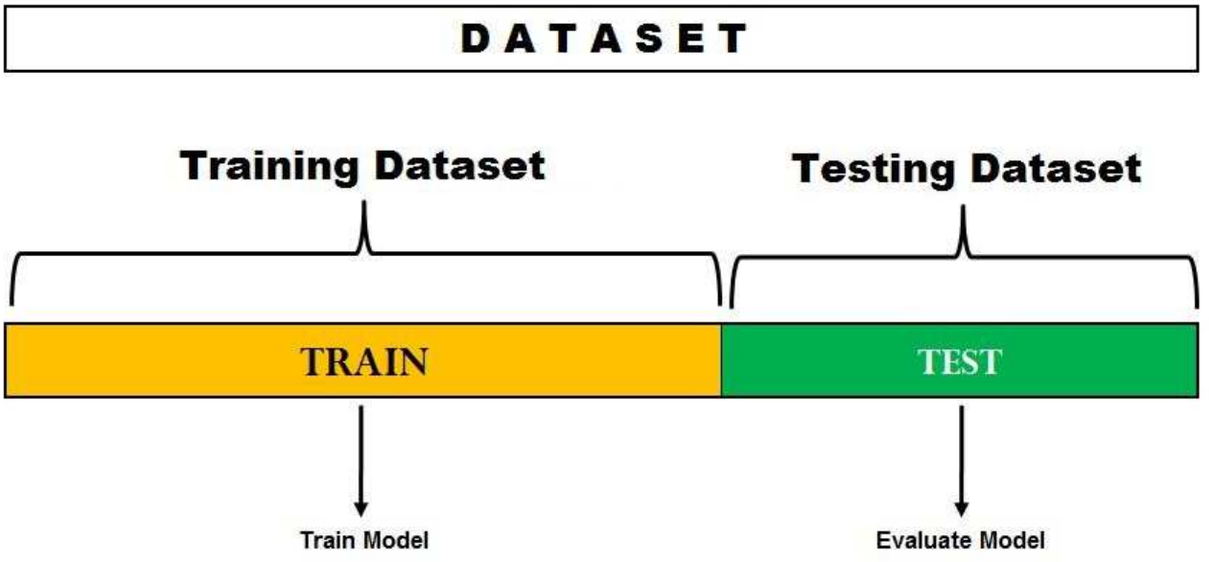



Image Source: DataVedas

The data split happens randomly, and we can't be sure which data ends up in the train and test bucket during the split unless we specify random_state. This can lead to extremely high variance and every time, the split changes, the accuracy will also change.

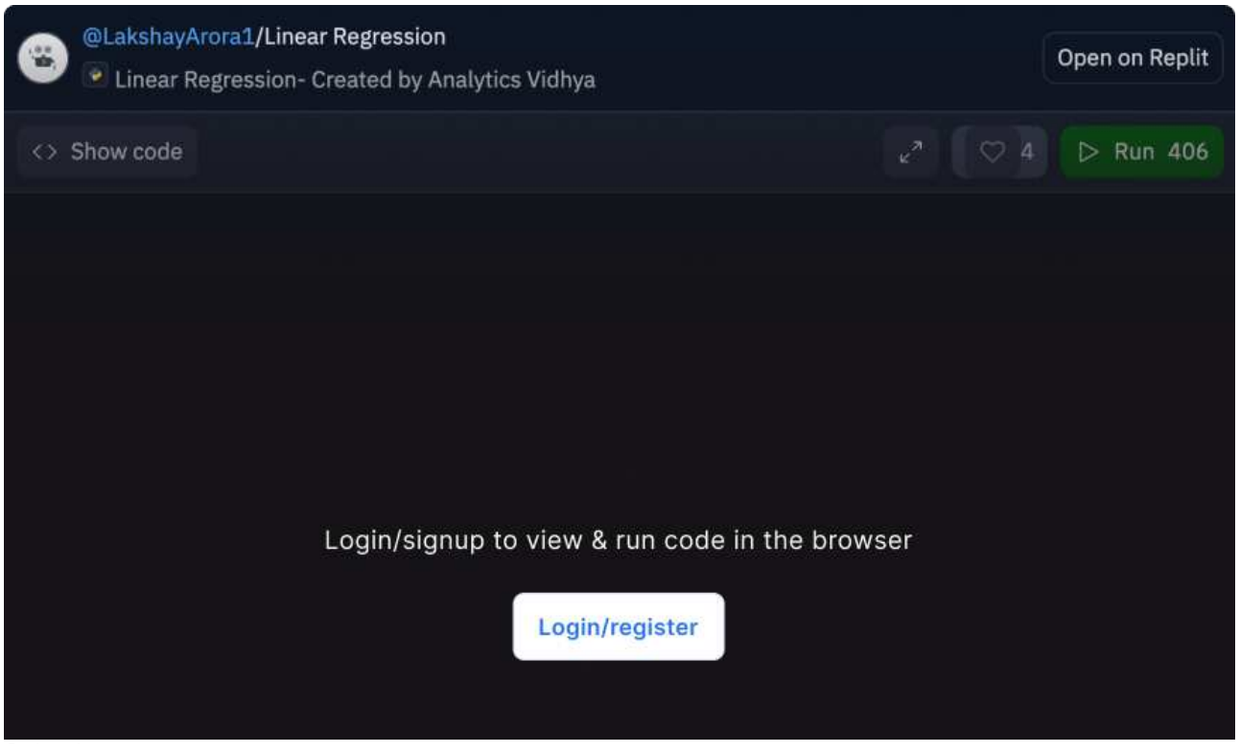
There are some drawbacks to this method:

- In the Hold out method, the test error rates are highly variable (**high variance**) and it totally depends on which observations end up in the training set and test set
- Only a part of the data is used to train the model (**high bias**) which is not a very good idea when data is not huge and this will lead to overestimation of test error.

One of the major advantages of this method is that it is computationally inexpensive compared to other cross-validation techniques.

**Quick implementation of Hold Out method in Python**

**Python Code:**




```
Output
Train: [50, 10, 40, 20, 80, 90, 60] Test: [30, 100, 70]
```

Here, *random_state* is the seed used for reproducibility.

## 2. Leave One Out Cross-Validation

In this method, we divide the data into train and test sets – but with a twist. Instead of dividing the data into 2 subsets, we select a single observation as test data, and everything else is labeled as training data and the model is trained. Now the 2nd observation is selected as test data and the model is trained on the remaining data.

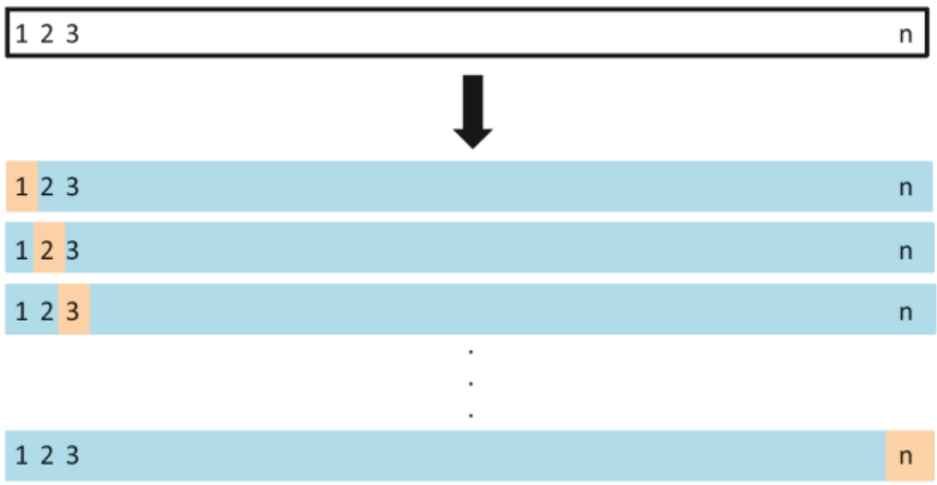



Image Source: ISLR

This process continues ‘n’ times and the average of all these iterations is calculated and estimated as the test set error.

$$\mathrm{CV}_{(n)} = \frac{1}{n}\sum_{i=1}^{n}\mathrm{MSE}_i.$$

When it comes to test-error estimates, LOOCV gives unbiased estimates (**low bias**). But bias is not the only matter of concern in estimation problems. We should also consider variance.

LOOCV has an extremely **high variance** because we are averaging the output of n-models which are fitted on an almost identical set of observations, and their outputs are highly positively correlated with each other.

And you can clearly see this is computationally expensive as the model is run 'n' times to test every observation in the data. Our next method will tackle this problem and give us a good balance between bias and variance.

**Quick implementation of Leave One Out Cross-Validation in Python**

```
from sklearn.model_selection import LeaveOneOut
X = [10,20,30,40,50,60,70,80,90,100]

l = LeaveOneOut()

for train, test in l.split(X):

    print("%s %s"% (train,test))
Output
[1 2 3 4 5 6 7 8 9] [0]
[0 2 3 4 5 6 7 8 9] [1]
[0 1 3 4 5 6 7 8 9] [2]
[0 1 2 4 5 6 7 8 9] [3]
[0 1 2 3 5 6 7 8 9] [4]
[0 1 2 3 4 6 7 8 9] [5]
[0 1 2 3 4 5 7 8 9] [6]
[0 1 2 3 4 5 6 8 9] [7]
[0 1 2 3 4 5 6 7 9] [8]
[0 1 2 3 4 5 6 7 8] [9]
```

This output clearly shows how LOOCV keeps one observation aside as test data and all the other observations go to train data.

# 3. K-Fold Cross-Validation

In this resampling technique, the whole data is divided into k sets of almost equal sizes. The first set is selected as the test set and the model is trained on the remaining k-1 sets. The test error rate is then calculated after fitting the model to the test data.

In the second iteration, the 2nd set is selected as a test set and the remaining k-1 sets are used to train the data and the error is calculated. This process continues for all the k sets.

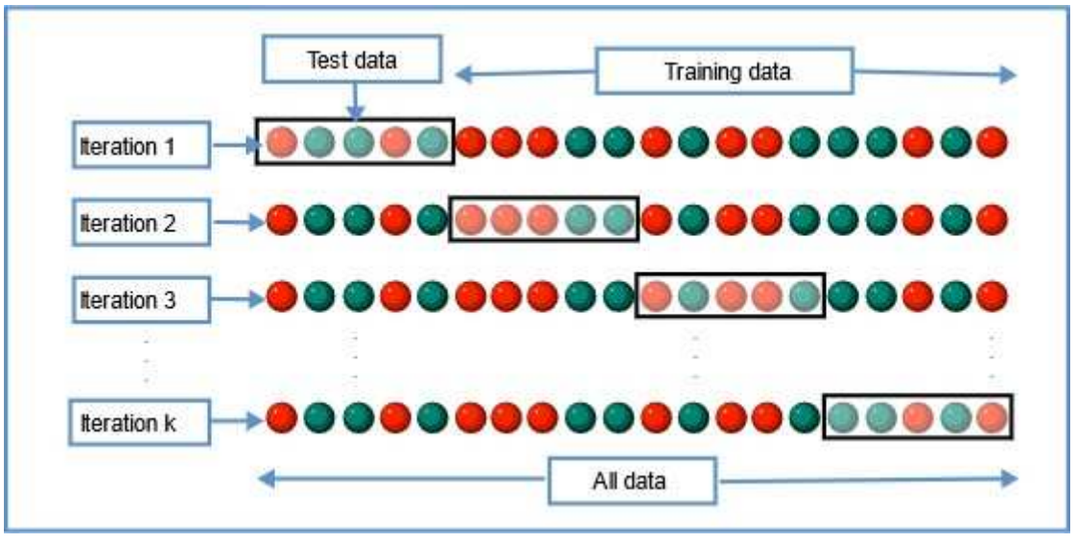



Image Source: Wikipedia

The mean of errors from all the iterations is calculated as the CV test error estimate.

$$\mathrm{CV}_{(k)} = \frac{1}{k}\sum_{i=1}^{k}\mathrm{MSE}_i.$$

In K-Fold CV, the no of folds k is less than the number of observations in the data (k<n) and we are averaging the outputs of k fitted models that are somewhat less correlated with each other since the overlap between the training sets in each model is smaller. This leads to **low variance** then LOOCV.

The best part about this method is each data point gets to be in the test set exactly once and gets to be part of the training set k-1 times. As the number of folds k increases, the variance also decreases (low variance). This method leads to **intermediate bias** because each training set contains fewer observations (k-1)n/k than the Leave One Out method but more than the Hold Out method.

Typically, K-fold Cross Validation is performed using k=5 or k=10 as these values have been empirically shown to yield test error estimates that neither have high bias nor high variance.

The major disadvantage of this method is that the model has to be run from scratch k-times and is computationally expensive than the Hold Out method but better than the Leave One Out method.

**Simple implementation of K-Fold Cross-Validation in Python**

```
from sklearn.model_selection import KFold
X = ["a",'b','c','d','e','f']

kf = KFold(n_splits=3, shuffle=False, random_state=None)

for train, test in kf.split(X):

    print("Train data",train,"Test data",test)
Output
Train: [2 3 4 5] Test: [0 1]
Train: [0 1 4 5] Test: [2 3]
Train: [0 1 2 3] Test: [4 5]
```

# 4. Stratified K-Fold Cross-Validation

This is a slight variation from K-Fold Cross Validation, which uses **'stratified sampling'** instead of 'random sampling.'

Let's quickly understand what stratified sampling is and how is it different from random sampling.

Suppose your data contains reviews for a cosmetic product used by both the male and female population. When we perform random sampling to split the data into train and test sets, there is a possibility that most of the data representing males is not represented in training data but might end up in test data. When we train the model on sample training data that is not a correct representation of the actual population, the model will not predict the test data with good accuracy.

This is where Stratified Sampling comes to the rescue. Here the data is split in such a way that it represents all the classes from the population.

Let's consider the above example which has a cosmetic product review of 1000 customers out of which 60% is female and 40% is male. I want to split the data into train and test data in proportion (80:20). 80% of 1000 customers will be 800 which will be chosen in such a way that there are 480 reviews associated with the female population and 320 representing the male population. In a similar fashion, 20% of 1000 customers will be chosen for the test data ( with the same female and male representation).

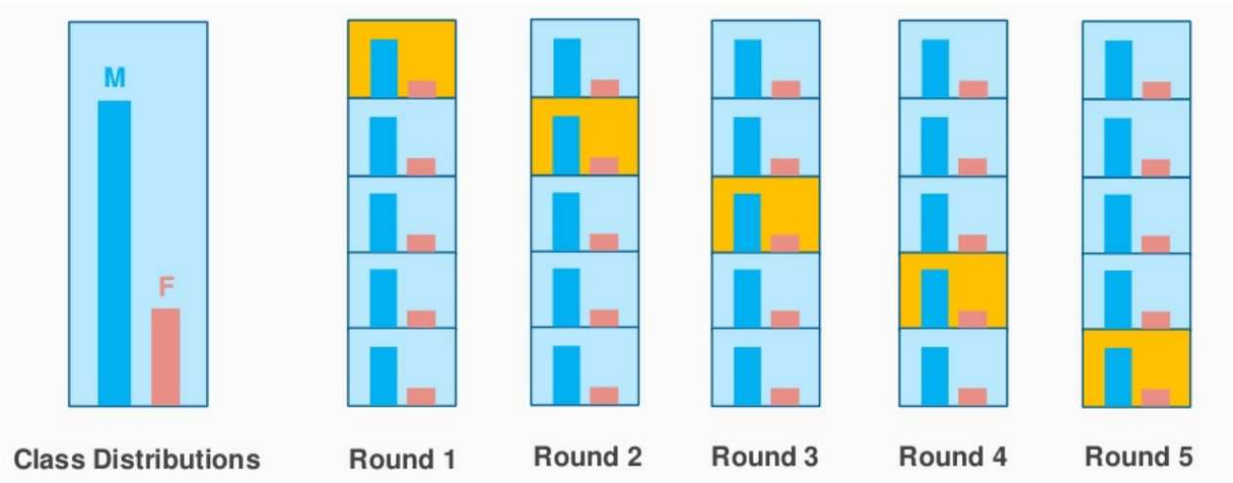



Image Source: stackexchange.com

This is exactly what stratified K-Fold CV does and it will create K-Folds by preserving the percentage of sample for each class. This solves the problem of random sampling associated with Hold out and K-Fold methods.

### Quick implementation of Stratified K-Fold Cross-Validation in Python

```
from sklearn.model_selection import StratifiedKFold
X = np.array([[1,2],[3,4],[5,6],[7,8],[9,10],[11,12]])
y= np.array([0,0,1,0,1,1])

skf = StratifiedKFold(n_splits=3,random_state=None,shuffle=False)

for train_index,test_index in skf.split(X,y):
    print("Train:",train_index,'Test:',test_index)
    X_train,X_test = X[train_index], X[test_index]
    y_train,y_test = y[train_index], y[test_index]
Output
Train: [1 3 4 5] Test: [0 2]
Train: [0 2 3 5] Test: [1 4]
Train: [0 1 2 4] Test: [3 5]
```

The output clearly shows the stratified split done based on the classes ‘0’ and ‘1’ in ‘y’.

## Bias – Variance Tradeoff

When we consider the test error rate estimates, K-Fold Cross Validation gives more accurate estimates than Leave One Out Cross-Validation. Whereas Hold One Out CV method usually leads to overestimates of the test error rate, because in this approach, only a portion of the data is used to train the machine learning model.

When it comes to bias, the Leave One Out Method gives unbiased estimates because each training set contains n-1 observations (which is pretty much all of the data). K-Fold CV leads to an intermediate level of bias depending on the number of k-folds when compared to LOOCV but it’s much lower when compared to the Hold Out Method.

To conclude, the Cross-Validation technique that we choose highly depends on the use case and bias-variance trade-off.

If you have read this article this far, here is a quick bonus for you 👏

**sklearn.model_selection** has a method **cross_val_score** which simplifies the process of cross-validation. Instead of iterating through the complete data using the ‘split’ function, we can use *cross_val_score* and check the accuracy score for the chosen cross-validation method

You can check out my Github for python implementation of different cross-validation methods on the UCI Breast Cancer data from Kaggle.

## Frequently Asked Questions

**Q1. What is cross-validation and its types?**

A. Cross-validation is a technique used in machine learning and statistical modeling to assess the performance of a model and to prevent overfitting. It involves dividing the dataset into multiple subsets, using some for training the model and the rest for testing, multiple times to obtain reliable performance metrics.

Types of Cross-validation:
1. **K-Fold Cross-Validation**: The data is divided into K subsets (or “folds”). The model is trained K times, using K-1 folds for training and one fold for testing in each iteration.
2. **Leave-One-Out Cross-Validation (LOOCV)**: K-Fold CV with K equal to the number of data points, i.e., each data point is used once as a test set, and the model is trained K times.
3. **Stratified K-Fold Cross-Validation**: It ensures that the class distribution remains similar in each fold, important when dealing with imbalanced datasets.
4. **Time Series Cross-Validation**: For time-dependent data, it uses a series of temporally ordered training and testing sets, preventing the use of future data for training.
5. **Shuffle-Split Cross-Validation**: Randomly shuffles the data, and then splits it into training and testing sets multiple times.
6. **Group K-Fold Cross-Validation**: Useful when the data contains groups, like multiple samples from the same subject, and ensures all samples from a group are kept together in the same fold.

Cross-validation provides a more robust assessment of a model’s performance and helps in selecting hyperparameters, assessing generalization, and avoiding overfitting by evaluating the model on different data points than those used for training.

**Q2. What is K-fold used for?**

A. K-fold cross-validation is used to assess the performance and generalization of a machine learning model. It divides the dataset into K subsets, trains the model K times using K-1 folds for

training and one fold for testing in each iteration. This method helps in obtaining more reliable performance metrics, choosing hyperparameters, and preventing overfitting by evaluating the model on different data subsets.

The End