

1 Introduction and Overview

How is it that a committee of blockheads can somehow arrive at highly reasoned decisions, despite the weak judgment of the individual members? How can the shaky separate views of a panel of dolts be combined into a single opinion that is very likely to be correct? That this possibility of garnering wisdom from a council of fools can be harnessed and used to advantage may seem far-fetched and implausible, especially in real life. Nevertheless, this unlikely strategy turns out to form the basis of *boosting*, an approach to machine learning that is the topic of this book. Indeed, at its core, boosting solves hard machine-learning problems by forming a very smart committee of grossly incompetent but carefully selected members.

To see how this might work in the context of machine learning, consider the problem of filtering out spam, or junk email. Spam is a modern-day nuisance, and one that is ideally handled by highly accurate filters that can identify and remove spam from the flow of legitimate email. Thus, to build a spam filter, the main problem is to create a method by which a computer can automatically categorize email as spam (junk) or ham (legitimate). The machine learning approach to this problem prescribes that we begin by gathering a collection of examples of the two classes, that is, a collection of email messages which have been labeled, presumably by a human, as spam or ham. The purpose of the machine learning algorithm is to automatically produce from such data a prediction rule that can be used to reliably classify new examples (email messages) as spam or ham.

For any of us who has ever been bombarded with spam, rules for identifying spam or ham will immediately come to mind. For instance, if it contains the word *Viagra*, then it is probably spam. Or, as another example, email from one's spouse is quite likely to be ham. Such individual rules of thumb are far from complete as a means of separating spam from ham. A rule that classifies all email containing *Viagra* as spam, and all other email as ham, will very often be wrong. On the other hand, such a rule is undoubtedly telling us something useful and nontrivial, and its accuracy, however poor, will nonetheless be significantly better than simply guessing entirely at random as to whether each email is spam or ham.

Intuitively, finding these weak rules of thumb should be relatively easy—so easy, in fact, that one might reasonably envision a kind of automatic “weak learning” program that,

given any set of email examples, could effectively search for a simple prediction rule that may be rough and rather inaccurate, but that nonetheless provides some nontrivial guidance in separating the given examples as spam or ham. Furthermore, by calling such a weak learning program repeatedly on various subsets of our dataset, it would be possible to extract a collection of rules of thumb. The main idea of boosting is to somehow combine these weak and inaccurate rules of thumb into a single “committee” whose overall predictions will be quite accurate.

In order to use these rules of thumb to maximum advantage, there are two critical problems that we face: First, how should we choose the collections of email examples presented to the weak learning program so as to extract rules of thumb that will be the most useful? And second, once we have collected many rules of thumb, how can they be combined into a single, highly accurate prediction rule? For the latter question, a reasonable approach is simply for the combined rule to take a vote of the predictions of the rules of thumb. For the first question, we will advocate an approach in which the weak learning program is forced to focus its attention on the “hardest” examples, that is, the ones for which the previously chosen rules of thumb were most apt to give incorrect predictions.

Boosting refers to a general and provably effective method of producing a very accurate prediction rule by combining rough and moderately inaccurate rules of thumb in a manner similar to that suggested above. This book presents in detail much of the recent work on boosting, focusing especially on the *AdaBoost* algorithm, which has undergone intense theoretical study and empirical testing. In this first chapter, we introduce *AdaBoost* and some of the key concepts required for its study. We also give a brief overview of the entire book.

See the appendix for a description of the notation used here and throughout the book, as well as some brief, mathematical background.

1.1 Classification Problems and Machine Learning

This book focuses primarily on *classification* problems in which the goal is to categorize objects into one of a relatively small set of classes. For instance, an optical character recognition (OCR) system must classify images of letters into the categories *A*, *B*, *C*, etc. Medical diagnosis is another example of a classification problem in which the goal is to diagnose a patient. In other words, given the symptoms manifested by the patient, our goal is to categorize him or her as a sufferer or non-sufferer of a particular disease. The spam-filtering example is also a classification problem in which we attempt to categorize emails as spam or ham.

We focus especially on a machine-learning approach to classification problems. Machine learning studies the design of automatic methods for making predictions about the future based on past experiences. In the context of classification problems, machine-learning methods attempt to learn to predict the correct classifications of unseen examples through the careful examination of examples which were previously labeled with their correct classifications, usually by a human.

We refer to the objects to be classified as *instances*. Thus, an instance is a description of some kind which is used to derive a predicted classification. In the OCR example, the instances are the images of letters. In the medical-diagnosis example, the instances are descriptions of a patient's symptoms. The space of all possible instances is called the *instance space* or *domain*, and is denoted by \mathcal{X} . A (*labeled*) *example* is an instance together with an associated *label* indicating its correct classification. Instances are also sometimes referred to as (unlabeled) examples.

During training, a *learning algorithm* receives as input a *training set* of labeled examples called the *training examples*. The output of the learning algorithm is a prediction rule called a *classifier* or *hypothesis*. A classifier can itself be thought of as a computer program which takes as input a new unlabeled instance and outputs a predicted classification; so, in mathematical terms, a classifier is a function that maps instances to labels. In this book, we use the terms *classifier* and *hypothesis* fairly interchangeably, with the former emphasizing a prediction rule's use in classifying new examples, and the latter emphasizing the fact that the rule has been (or could be) generated as the result of some learning process. Other terms that have been used in the literature include *rule*, *prediction rule*, *classification rule*, *predictor*, and *model*.

To assess the quality of a given classifier, we measure its error rate, that is, the frequency with which it makes incorrect classifications. To do this, we need a *test set*, a separate set of *test examples*. The classifier is evaluated on each of the test instances, and its predictions are compared against the correct classifications of the test examples. The fraction of examples on which incorrect classifications were made is called the *test error* of the classifier. Similarly, the fraction of mistakes on the training set is called the *training error*. The fraction of correct predictions is called the (test or training) *accuracy*.

Of course, the classifier's performance on the training set is not of much interest since our purpose is to build a classifier that works well on unseen data. On the other hand, if there is no relationship at all between the training set and the test set, then the learning problem is unsolvable; the future can be predicted only if it resembles the past. Therefore, in designing and studying learning algorithms, we generally assume that the training and test examples are taken from the *same* random source. That is, we assume that the examples are chosen randomly from some fixed but unknown distribution \mathcal{D} over the space of labeled examples and, moreover, that the training and test examples are generated by the *same* distribution. The *generalization error* of a classifier measures the probability of misclassifying a random example from this distribution \mathcal{D} ; equivalently, the generalization error is the expected test error of the classifier on any test set generated by \mathcal{D} . The goal of learning can now be stated succinctly as producing a classifier with low generalization error.

To illustrate these concepts, consider the problem of diagnosing a patient with coronary artery disease. For this problem, an instance consists of a description of the patient including items such as sex, age, cholesterol level, chest pain type (if any), blood pressure, and results of various medical tests. The label or class associated with each instance is a diagnosis provided by a doctor as to whether or not the patient described actually suffers from the

disease. During training, a learning algorithm is provided with a set of labeled examples and attempts to produce a classifier for predicting if new patients suffer from the disease. The goal is to produce a classifier that is as accurate as possible. Later, in section 1.2.3, we describe experiments using a publicly available dataset for this problem.

1.2 Boosting

We can now make some of the informal notions about boosting described above more precise. Boosting assumes the availability of a *base* or *weak learning algorithm* which, given labeled training examples, produces a *base* or *weak classifier*. The goal of boosting is to improve the performance of the weak learning algorithm while treating it as a “black box” which can be called repeatedly, like a subroutine, but whose innards cannot be observed or manipulated. We wish to make only the most minimal assumptions about this learning algorithm. Perhaps the least we can assume is that the weak classifiers are not entirely trivial in the sense that their error rates are at least a little bit better than a classifier whose every prediction is a random guess. Thus, like the rules of thumb in the spam-filtering example, the weak classifiers can be rough and moderately inaccurate, but not entirely trivial and uninformative. This assumption, that the base learner produces a weak hypothesis that is at least slightly better than random guessing on the examples on which it was trained, is called the *weak learning assumption*, and it is central to the study of boosting.

As with the words *classifier* and *hypothesis*, we use the terms *base* and *weak* roughly interchangeably, with *weak* emphasizing mediocrity in performance and *base* connoting use as a building block.

Like any learning algorithm, a boosting algorithm takes as input a set of training examples $(x_1, y_1), \dots, (x_m, y_m)$ where each x_i is an instance from \mathcal{X} , and each y_i is the associated label or class. For now, and indeed for most of this book, we assume the simplest case in which there are only two classes, -1 and $+1$, although we do explore extensions to multiclass problems in chapter 10.

A boosting algorithm’s only means of learning from the data is through calls to the base learning algorithm. However, if the base learner is simply called repeatedly, always with the same set of training data, we cannot expect anything interesting to happen; instead, we expect the same, or nearly the same, base classifier to be produced over and over again, so that little is gained over running the base learner just once. This shows that the boosting algorithm, if it is to improve on the base learner, must in some way manipulate the data that it feeds to it.

Indeed, the key idea behind boosting is to choose training sets for the base learner in such a fashion as to force it to infer something new about the data each time it is called. This can be accomplished by choosing training sets on which we can reasonably expect the performance of the preceding base classifiers to be very poor—even poorer than their regular weak performance. If this can be accomplished, then we can expect the base learner

to output a new base classifier which is significantly different from its predecessors. This is because, although we think of the base learner as a weak and mediocre learning algorithm, we nevertheless expect it to output classifiers that make nontrivial predictions.

We are now ready to describe in detail the boosting algorithm AdaBoost, which incorporates these ideas, and whose pseudocode is shown as algorithm 1.1. AdaBoost proceeds in *rounds* or iterative calls to the base learner. For choosing the training sets provided to the base learner on each round, AdaBoost maintains a *distribution* over the training examples. The distribution used on the t -th round is denoted D_t , and the weight it assigns to training example i is denoted $D_t(i)$. Intuitively, this weight is a measure of the importance of correctly classifying example i on the current round. Initially, all weights are set equally, but on each round, the weights of incorrectly classified examples are increased so that, effectively, hard examples get successively higher weight, forcing the base learner to focus its attention on them.

Algorithm 1.1

The boosting algorithm AdaBoost

Given: $(x_1, y_1), \dots, (x_m, y_m)$ where $x_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$.

Initialize: $D_1(i) = 1/m$ for $i = 1, \dots, m$.

For $t = 1, \dots, T$:

- Train weak learner using distribution D_t .
- Get weak hypothesis $h_t : \mathcal{X} \rightarrow \{-1, +1\}$.
- Aim: select h_t to minimize the weighted error:

$$\epsilon_t \doteq \Pr_{i \sim D_t}[h_t(x_i) \neq y_i].$$

- Choose $\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$.
- Update, for $i = 1, \dots, m$:

$$\begin{aligned} D_{t+1}(i) &= \frac{D_t(i)}{Z_t} \times \begin{cases} e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\ e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i \end{cases} \\ &= \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}, \end{aligned}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(x) \right).$$

The base learner's job is to find a base classifier $h_t : \mathcal{X} \rightarrow \{-1, +1\}$ appropriate for the distribution D_t . Consistent with the earlier discussion, the quality of a base classifier is measured by its error *weighted* by the distribution D_t :

$$\epsilon_t \doteq \Pr_{i \sim D_t}[h_t(x_i) \neq y_i] = \sum_{i: h_t(x_i) \neq y_i} D_t(i).$$

Here, $\Pr_{i \sim D_t}[\cdot]$ denotes probability with respect to the random selection of an example (as specified by its index i) according to distribution D_t . Thus, the weighted error ϵ_t is the chance of h_t misclassifying a random example if selected according to D_t . Equivalently, it is the sum of the weights of the misclassified examples. Notice that the error is measured with respect to the same distribution D_t on which the base classifier was trained.

The weak learner attempts to choose a weak hypothesis h_t with low weighted error ϵ_t . In this setting, however, we do not expect that this error will be especially small in an absolute sense, but only in a more general and relative sense; in particular, we expect it to be only a bit better than random, and typically far from zero. To emphasize this looseness in what we require of the weak learner, we say that the weak learner's aim is to *minimalize* the weighted error, using this word to signify a vaguer and less stringent diminishment than that connoted by *minimize*.

If a classifier makes each of its predictions entirely at random, choosing each predicted label to be -1 or $+1$ with equal probability, then its probability of misclassifying any given example will be exactly $\frac{1}{2}$. Therefore, the error of this classifier will always be $\frac{1}{2}$, regardless of the data on which the error is measured. Thus, a weak hypothesis with weighted error ϵ_t equal to $\frac{1}{2}$ can be obtained trivially by formulating each prediction as a random guess. The weak learning assumption then, for our present purposes, amounts to an assumption that the error of each weak classifier is bounded away from $\frac{1}{2}$, so that each ϵ_t is at most $\frac{1}{2} - \gamma$ for some small positive constant γ . In this way, each weak hypothesis is assumed to be slightly better than random guessing by some small amount, as measured by its error. (This assumption will be refined considerably in section 2.3.)

As for the weights $D_t(i)$ that AdaBoost calculates on the training examples, in practice, there are several ways in which these can be used by the base learner. In some cases, the base learner can use these weights directly. In other cases, an unweighted training set is generated for the base learner by selecting examples at random from the original training set. The probability of selecting an example in this case is set to be proportional to the weight of the example. These methods are discussed in more detail in section 3.4.

Returning to the spam-filtering example, the instances x_i correspond to email messages, and the labels y_i give the correct classification as spam or ham. The base classifiers are the rules of thumb provided by the weak learning program where the subcollections on which it is run are chosen randomly according to the distribution D_t .

Once the base classifier h_t has been received, AdaBoost chooses a parameter α_t as in algorithm 1.1. Intuitively, α_t measures the importance that is assigned to h_t . The precise

choice of α_t is unimportant for our present purposes; the rationale for this particular choice will become apparent in chapter 3. For now, it is enough to observe that $\alpha_t > 0$ if $\epsilon_t < \frac{1}{2}$, and that α_t gets larger as ϵ_t gets smaller. Thus, the more accurate the base classifier h_t , the more importance we assign to it.

The distribution D_t is next updated using the rule shown in the algorithm. First, all of the weights are multiplied either by $e^{-\alpha_t} < 1$ for examples correctly classified by h_t , or by $e^{\alpha_t} > 1$ for incorrectly classified examples. Equivalently, since we are using labels and predictions in $\{-1, +1\}$, this update can be expressed more succinctly as a scaling of each example i by $\exp(-\alpha_t y_i h_t(x_i))$. Next, the resulting set of values is renormalized by dividing through by the factor Z_t to ensure that the new distribution D_{t+1} does indeed sum to 1. The effect of this rule is to increase the weights of examples misclassified by h_t , and to decrease the weights of correctly classified examples. Thus, the weight tends to concentrate on “hard” examples. Actually, to be more precise, AdaBoost chooses a new distribution D_{t+1} on which the last base classifier h_t is sure to do extremely poorly: It can be shown by a simple computation that the error of h_t with respect to distribution D_{t+1} is exactly $\frac{1}{2}$, that is, exactly the trivial error rate achievable through simple random guessing (see exercise 1.1). In this way, as discussed above, AdaBoost tries on each round to force the base learner to learn something new about the data.

After many calls to the base learner, AdaBoost combines the many base classifiers into a single *combined* or *final classifier* H . This is accomplished by a simple weighted vote of the base classifiers. That is, given a new instance x , the combined classifier evaluates all of the base classifiers, and predicts with the weighted majority of the base classifiers’ predicted classifications. Here, the vote of the t -th base classifier h_t is weighted by the previously chosen parameter α_t . The resulting formula for H ’s prediction is as shown in the algorithm.

1.2.1 A Toy Example

To illustrate how AdaBoost works, let us look at the tiny toy learning problem shown in figure 1.1. Here, the instances are points in the plane which are labeled $+$ or $-$. In this case, there are $m = 10$ training examples, as shown in the figure; five are positive and five are negative.

Let us suppose that our base learner finds classifiers defined by vertical or horizontal lines through the plane. For instance, such a base classifier defined by a vertical line might classify all points to the right of the line as positive, and all points to the left as negative. It can be checked that no base classifier of this form correctly classifies more than seven of the ten training examples, meaning that none has an unweighted training error below 30%. On each round t , we suppose that the base learner always finds the base hypothesis of this form that has *minimum* weighted error with respect to the distribution D_t (breaking ties arbitrarily). We will see in this example how, using such a base learner for finding such weak base classifiers, AdaBoost is able to construct a combined classifier that correctly classifies all of the training examples in only $T = 3$ boosting rounds.

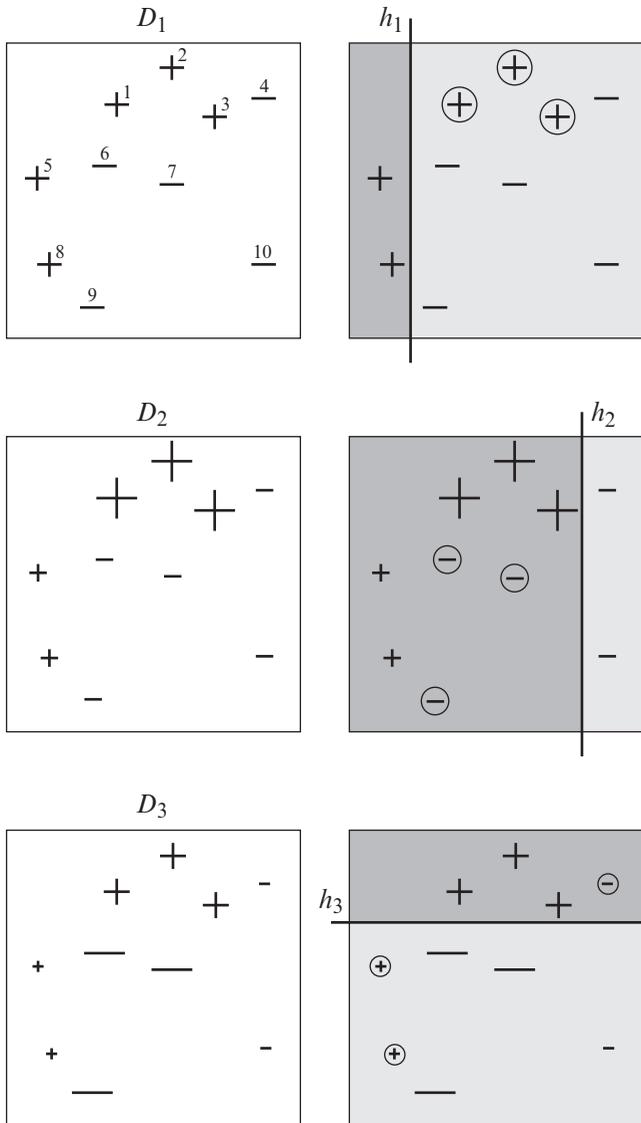


Figure 1.1

An illustration of how AdaBoost behaves on a tiny toy problem with $m = 10$ examples. Each row depicts one round, for $t = 1, 2, 3$. The left box in each row represents the distribution D_t , with the size of each example scaled in proportion to its weight under that distribution. Each box on the right shows the weak hypothesis h_t , where darker shading indicates the region of the domain predicted to be positive. Examples that are misclassified by h_t have been circled.

Table 1.1

The numerical calculations corresponding to the toy example in figure 1.1

	1	2	3	4	5	6	7	8	9	10	
$D_1(i)$	<u>0.10</u>	<u>0.10</u>	<u>0.10</u>	0.10	0.10	0.10	0.10	0.10	0.10	0.10	$\epsilon_1 = 0.30, \alpha_1 \approx 0.42$
$e^{-\alpha_1 y_i h_1(x_i)}$	1.53	1.53	1.53	0.65	0.65	0.65	0.65	0.65	0.65	0.65	
$D_1(i) e^{-\alpha_1 y_i h_1(x_i)}$	0.15	0.15	0.15	0.07	0.07	0.07	0.07	0.07	0.07	0.07	$Z_1 \approx 0.92$
$D_2(i)$	0.17	0.17	0.17	0.07	0.07	<u>0.07</u>	<u>0.07</u>	0.07	<u>0.07</u>	0.07	$\epsilon_2 \approx 0.21, \alpha_2 \approx 0.65$
$e^{-\alpha_2 y_i h_2(x_i)}$	0.52	0.52	0.52	0.52	0.52	1.91	1.91	0.52	1.91	0.52	
$D_2(i) e^{-\alpha_2 y_i h_2(x_i)}$	0.09	0.09	0.09	0.04	0.04	0.14	0.14	0.04	0.14	0.04	$Z_2 \approx 0.82$
$D_3(i)$	0.11	0.11	0.11	<u>0.05</u>	<u>0.05</u>	0.17	0.17	<u>0.05</u>	0.17	0.05	$\epsilon_3 \approx 0.14, \alpha_3 \approx 0.92$
$e^{-\alpha_3 y_i h_3(x_i)}$	0.40	0.40	0.40	2.52	2.52	0.40	0.40	2.52	0.40	0.40	
$D_3(i) e^{-\alpha_3 y_i h_3(x_i)}$	0.04	0.04	0.04	0.11	0.11	0.07	0.07	0.11	0.07	0.02	$Z_3 \approx 0.69$

Calculations are shown for the ten examples as numbered in the figure. Examples on which hypothesis h_t makes a mistake are indicated by underlined figures in the rows marked D_t .

On round 1, AdaBoost assigns equal weight to all of the examples, as is indicated in the figure by drawing all examples in the box marked D_1 to be of the same size. Given examples with these weights, the base learner chooses the base hypothesis indicated by h_1 in the figure, which classifies points as positive if and only if they lie to the left of this line. This hypothesis incorrectly classifies three points—namely, the three circled positive points—so its error ϵ_1 is 0.30. Plugging into the formula of algorithm 1.1 gives $\alpha_1 \approx 0.42$.

In constructing D_2 , the weights of the three points misclassified by h_1 are increased while the weights of all other points are decreased. This is indicated by the sizes of the points in the box marked D_2 . See also table 1.1, which shows the numerical calculations involved in running AdaBoost on this toy example.

On round 2, the base learner chooses the line marked h_2 . This base classifier correctly classifies the three relatively high-weight points missed by h_1 , though at the expense of missing three other comparatively low-weight points which were correctly classified by h_1 . Under distribution D_2 , these three points have weight only around 0.07, so the error of h_2 with respect to D_2 is $\epsilon_2 \approx 0.21$, giving $\alpha_2 \approx 0.65$. In constructing D_3 , the weights of these three misclassified points are increased while the weights of the other points are decreased.

On round 3, classifier h_3 is chosen. This classifier misses none of the points misclassified by h_1 and h_2 since these points have relatively high weight under D_3 . Instead, it misclassifies three points which, because they were not misclassified by h_1 or h_2 , are of very low weight under D_3 . On round 3, $\epsilon_3 \approx 0.14$ and $\alpha_3 \approx 0.92$.

Note that our earlier remark that the error of each hypothesis h_t is exactly $\frac{1}{2}$ on the new distribution D_{t+1} can be verified numerically in this case from table 1.1 (modulo small discrepancies due to rounding).

The combined classifier H is a weighted vote of h_1 , h_2 , and h_3 as shown in figure 1.2, where the weights on the respective classifiers are α_1 , α_2 , and α_3 . Although each of the composite weak classifiers misclassifies three of the ten examples, the combined classifier,

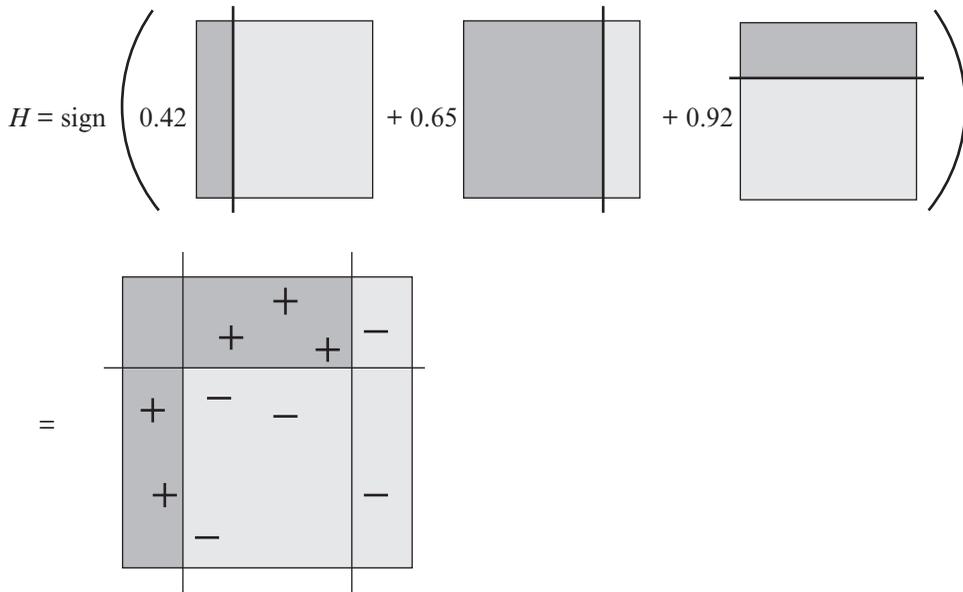


Figure 1.2

The combined classifier for the toy example of figure 1.1 is computed as the sign of the weighted sum of the three weak hypotheses, $\alpha_1 h_1 + \alpha_2 h_2 + \alpha_3 h_3$, as shown at the top. This is equivalent to the classifier shown at the bottom. (As in figure 1.1, the regions that a classifier predicts positive are indicated using darker shading.)

as shown in the figure, correctly classifies *all* of the training examples. For instance, the classification of the negative example in the upper right corner (instance #4), which is classified negative by h_1 and h_2 but positive by h_3 , is

$$\text{sign}(-\alpha_1 - \alpha_2 + \alpha_3) = \text{sign}(-0.15) = -1.$$

One might reasonably ask if such a rapid reduction in training error is typical for AdaBoost. The answer turns out to be yes in the following sense: Given the weak learning assumption (that is, that the error of each weak classifier ϵ_t is at most $\frac{1}{2} - \gamma$ for some $\gamma > 0$), we can prove that the training error of the combined classifier drops exponentially fast as a function of the number of weak classifiers combined. Although this fact, which is proved in chapter 3, says nothing directly about generalization error, it does suggest that boosting, which is so effective at driving down the training error, may also be effective at producing a combined classifier with low generalization error. And indeed, in chapter 4 and 5, we prove various theorems about the generalization error of AdaBoost's combined classifier.

Note also that although we depend on the weak learning assumption to prove these results, AdaBoost does not need to know the “edge” γ referred to above, but rather adjusts and adapts to errors ϵ_t , which may vary considerably, reflecting the varying levels of performance among

the base classifiers. It is in this sense that AdaBoost is an *adaptive boosting* algorithm—which is exactly what the name stands for.¹ Moreover, this adaptiveness is one of the key qualities that make AdaBoost practical.

1.2.2 Experimental Performance

Experimentally, on data arising from many real-world applications, AdaBoost also turns out to be highly effective. To get a sense of AdaBoost’s performance overall, we can compare it with other methods on a broad variety of publicly available benchmark datasets, an important methodology in machine learning since different algorithms can exhibit relative strengths that vary substantially from one dataset to the next. Here, we consider two base learning algorithms: one that produces quite weak and simple base classifiers called decision stumps; and the other, called C4.5, that is an established and already highly effective program for learning decision trees, which are generally more complex but also quite a bit more accurate than decision stumps. Both of these base classifiers are described further in sections 1.2.3 and 1.3.

Boosting algorithms work by improving the accuracy of the base learning algorithm. Figure 1.3 shows this effect on 27 benchmark datasets. In each scatterplot, each point shows the test error rate of boosting (x -coordinate) versus that of the base learner (y -coordinate) on a single benchmark. All error rates have been averaged over multiple runs and multiple random splits of the given data into training and testing sets. In these experiments, boosting was run for $T = 100$ rounds.

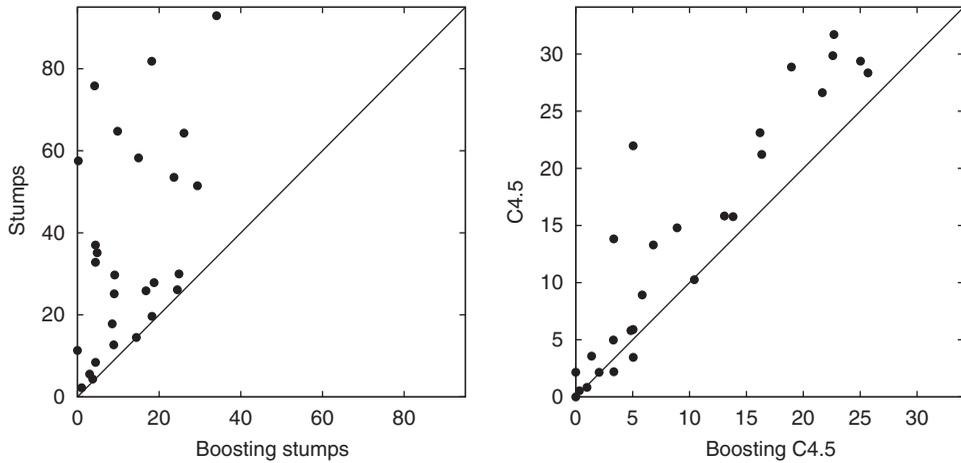
To “read” such a scatterplot, note that a point lands above the line $y = x$ if and only if boosting shows improvement over the base learner. Thus, we see that when using the relatively strong base learner C4.5, an algorithm that is very effective in its own right, AdaBoost is often able to provide quite a significant boost in performance. Even more dramatic is the improvement effected when using the rather weak decision stumps as base classifiers. In fact, this improvement is so substantial that boosting stumps is often even better than C4.5, as can be seen in figure 1.4. On the other hand, overall, boosting C4.5 seems to give more accurate results than boosting stumps.

In short, empirically, AdaBoost appears to be highly effective as a learning tool for generalizing beyond the training set. How can we explain this capacity to extrapolate beyond the observed training data? Attempting to answer this question is a primary objective of this book.

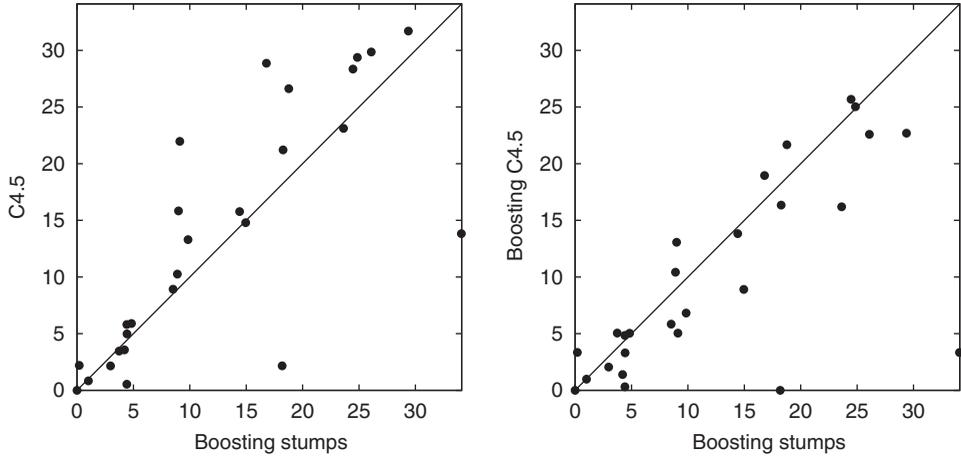
1.2.3 A Medical-Diagnosis Example

As a more detailed example, let us return to the heart-disease dataset described briefly in section 1.1. To apply boosting on this dataset, we first need to choose the base learner and base

1. This is also why *AdaBoost*, which is short for “adaptive boosting,” is pronounced *ADD-uh-boost*, similar to *adaptation*.

**Figure 1.3**

Comparison of two base learning algorithms—decision stumps and C4.5—with and without boosting. Each point in each scatterplot shows the test error rate of the two competing algorithms on one of 27 benchmark learning problems. The x -coordinate of each point gives the test error rate (in percent) using boosting, and the y -coordinate gives the error rate without boosting when using decision stumps (left plot) or C4.5 (right plot). All error rates have been averaged over multiple runs.

**Figure 1.4**

Comparison of boosting using decision stumps as the base learner versus unboosted C4.5 (left plot) and boosted C4.5 (right plot).

classifiers. Here we have many options, but perhaps the simplest rules of thumb are those which test on a single attribute describing the patient. For instance, such a rule might state:

If the patient's cholesterol is at least 228.5, then predict that the patient has heart disease; otherwise, predict that the patient is healthy.

In the experiments we are about to describe, we used base classifiers of just this form, which are the *decision stumps* alluded to in section 1.2.2. (In fact, the weak classifiers used in the toy example of section 1.2.1 are also decision stumps.) It turns out, as will be seen in section 3.4.2, that a base learner which does an exhaustive search for the best decision stump can be implemented very efficiently (where, as before, “best” means the one having lowest weighted training error with respect to a given distribution D_t over training examples). Table 1.2 shows the first six base classifiers produced by this base learner when AdaBoost is applied to this entire dataset.

To measure performance on such a small dataset, we can divide the data randomly into disjoint training and test sets. Because the test set for such a split is very small, we repeat this many times, using a standard technique called cross validation. We then take the averages of the training and test errors for the various splits of the data. Figure 1.5 shows these average error rates for this dataset as a function of the number of base classifiers combined. Boosting steadily drives down the training error. The test error also drops quickly, reaching a low point of 15.3% after only three rounds, a rather significant improvement over using just one of the base classifiers, the best of which has a test error of 28.0%. However, after reaching this low point, the test error begins to *increase* again, so that after 100 rounds, the test error is up to 18.8%, and after 1000 rounds, up to 22.0%.

This deterioration in performance with continued training is an example of an important and ubiquitous phenomenon called *overfitting*. As the number of base classifiers becomes larger and larger, the combined classifier becomes more and more complex, leading somehow to a deterioration of test-error performance. Overfitting, which has been observed in many machine-learning settings and which has also received considerable theoretical study, is consistent with the intuition that a simpler explanation of the data is better than a more

Table 1.2

The first six base classifiers found when using AdaBoost on the heart-disease dataset

Round	If	Then Predict	Else Predict
1	thalamus normal	healthy	sick
2	number of major vessels colored by fluoroscopy > 0	sick	healthy
3	chest pain type is asymptomatic	sick	healthy
4	ST depression induced by exercise relative to rest ≥ 0.75	sick	healthy
5	cholesterol ≥ 228.5	sick	healthy
6	resting electrocardiographic results are normal	healthy	sick

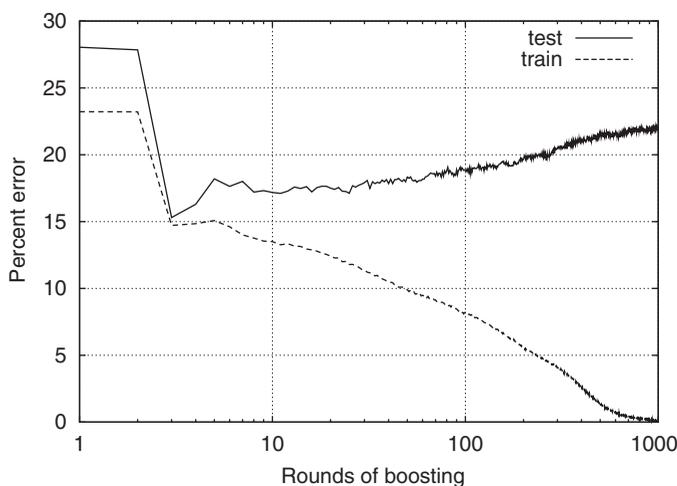


Figure 1.5

The training and test percent error rates obtained using boosting on the heart-disease dataset. Results are averaged over multiple train-test splits of the data.

complicated one, a notion sometimes called “Occam’s razor.” With more rounds of boosting, the combined classifier grows in size and complexity, apparently overwhelming good performance on the training set. This general connection between simplicity and accuracy is explored in chapter 2. For boosting, exactly the kind of behavior observed in figure 1.5 is predicted by the analysis in chapter 4.

Overfitting is a significant problem because it means that we have to be very careful about when to stop boosting. If we stop too soon or too late, our performance on the test set may suffer significantly, as can be seen in this example. Moreover, performance on the training set provides little guidance about when to stop training since the training error typically continues to drop even as overfitting gets worse and worse.

1.3 Resistance to Overfitting and the Margins Theory

This last example describes a case in which boosting was used with very weak base classifiers. This is one possible use of boosting, namely, in conjunction with a very simple but truly mediocre weak learning algorithm. A rather different use of boosting is instead to boost the accuracy of a learning algorithm that is already quite good.

This is the approach taken in the next example. Here, rather than a very weak base learner, we used the well-known and highly developed machine-learning algorithm C4.5 as the base learner. As mentioned earlier, C4.5 produces classifiers called *decision trees*. Figure 1.6 shows an example of a decision tree. The nodes are identified with tests having a small

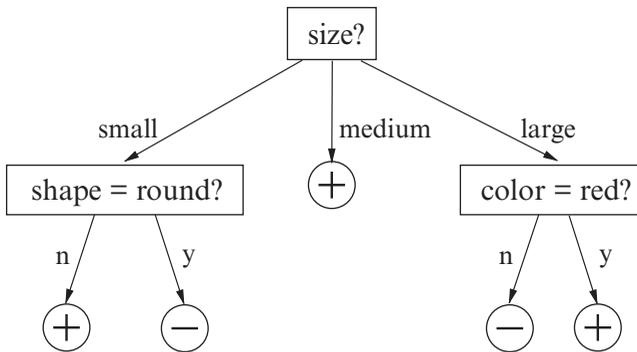


Figure 1.6
An example of a decision tree.

number of outcomes corresponding to the outgoing edges of the node. The leaves are identified with predicted labels. To classify an example, a path is traversed through the tree from the root to a leaf. The path is determined by the outcomes of the tests that are encountered along the way, and the predicted classification is determined by the leaf that is ultimately reached. For instance, in the figure, a large, square, blue item would be classified $-$ while a medium, round, red item would be classified $+$.

We tested boosting using C4.5 as the base learner on a benchmark dataset in which the goal is to identify images of handwritten characters as letters of the alphabet. The features used are derived from the raw pixel images, including such items as the average of the x -coordinates of the pixels that are turned on. The dataset consists of 16,000 training examples and 4000 test examples.

Figure 1.7 shows training and test error rates for AdaBoost's combined classifier on this dataset as a function of the number of decision trees (base classifiers) combined. A single decision tree produced by C4.5 on this dataset has a test error rate of 13.8%. In this example, boosting very quickly drives down the training error; in fact, after only five rounds the training error is zero, so that all training examples are correctly classified. Note that there is no reason why boosting cannot proceed beyond this point. Although the training error of the *combined* classifier is zero, the individual *base* classifiers continue to incur significant weighted error—around 5–6%—on the distributions on which they are trained, so that ϵ_t remains in this range, even for large t . This permits AdaBoost to proceed with the reweighting of training examples and the continued training of base classifiers.

The test performance of boosting on this dataset is extremely good, far better than a single decision tree. And surprisingly, unlike the earlier example, the test error on this dataset never increases, even after 1000 trees have been combined—by which point, the combined classifier involves more than two million decision nodes. Even after the training error hits zero, the test error continues to drop, from 8.4% on round 5 down to 3.1% on round 1000.

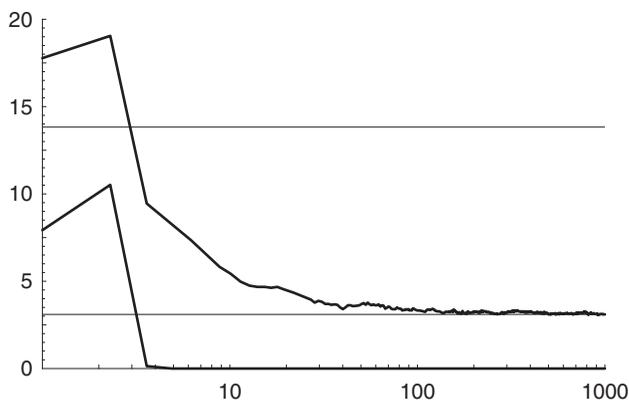


Figure 1.7

The training and test percent error rates obtained using boosting on an OCR dataset with C4.5 as the base learner. The top and bottom curves are test and training error, respectively. The top horizontal line shows the test error rate using just C4.5. The bottom line shows the final test error rate of AdaBoost after 1000 rounds. (Reprinted with permission of the Institute of Mathematical Statistics.)

This pronounced lack of overfitting seems to flatly contradict our earlier intuition that simpler is better. Surely, a combination of five trees is much, much simpler than a combination of 1000 trees (about 200 times simpler, in terms of raw size), and both perform equally well on the training set (perfectly, in fact). So how can it be that the far larger and more complex combined classifier performs so much better on the test set? This would appear to be a paradox.

One superficially plausible explanation is that the α_t 's are converging rapidly to zero, so that the number of base classifiers being combined is effectively bounded. However, as noted above, the ϵ_t 's remain around 5–6% in this case, well below $\frac{1}{2}$, which means that the weights α_t on the individual base classifiers are also bounded well above zero, so that the combined classifier is constantly growing and evolving with each round of boosting.

Such resistance to overfitting is typical of boosting, although, as we have seen in section 1.2.3, boosting certainly *can* overfit. This resistance is one of the properties that make it such an attractive learning algorithm. But how can we understand this behavior?

In chapter 5, we present a theoretical explanation of how, why, and when AdaBoost works and, in particular, of why it often does not overfit. Briefly, the main idea is the following. The description above of AdaBoost's performance on the training set took into account only the training error, which is already zero after just five rounds. However, training error tells only part of the story, in that it reports just the number of examples that are correctly or incorrectly classified. Instead, to understand AdaBoost, we also need to consider how *confident* the predictions being made by the algorithm are. We will see that such confidence can be measured by a quantity called the *margin*. According to this explanation, although the training error—that is, whether or not the predictions are correct—is not changing

after round 5, the confidence in those predictions is increasing dramatically with additional rounds of boosting. And it is this increase in confidence which accounts for the better generalization performance.

This theory, for which we present both empirical and theoretical evidence, not only explains the lack of overfitting but also provides a detailed framework for fundamentally understanding the conditions under which AdaBoost can fail or succeed.

1.4 Foundations and Algorithms

The core analysis outlined above forms part I of this book, a largely mathematical study of AdaBoost's capacity to minimize both the training and the generalization error. Here, our focus is on understanding how, why, and when AdaBoost is effective as a learning algorithm.

This analysis, including the margins theory, is paramount in our study of boosting; however, it is hardly the end of the story. Indeed, although it is an enticingly simple algorithm, AdaBoost turns out to be understandable from a striking number of disparate theoretical perspectives. Taken together, these provide a remarkably rich and encompassing illumination of the algorithm, in addition to practical generalizations and variations along multiple dimensions. Part II of the book explores three of these fundamental perspectives.

In the first of these, the interaction between a boosting algorithm and a weak learning algorithm is viewed as a game between these two players—a game not only in the informal, everyday sense but also in the mathematical sense studied in the field of game theory. In fact, it turns out that AdaBoost is a special case of a more general algorithm for playing any game in a repeated fashion. This perspective, presented in chapter 6, helps us to understand numerous properties of the algorithm, such as its limiting behavior, in broader, game-theoretic terms. We will see that notions that are central to boosting, such as margins and the weak learning assumption, have very natural game-theoretic interpretations. Indeed, the very idea of boosting turns out to be intimately entwined with one of the most fundamental theorems of game theory. This view also unifies AdaBoost with another branch of learning known as online learning.

AdaBoost can be further understood as an algorithm for optimizing a particular objective function measuring the fit of a model to the available data. In this way, AdaBoost can be seen as an instance of a more general approach that can be applied to a broader range of statistical learning problems, as we describe in chapter 7. This view further leads to a unification of AdaBoost with the more established statistical method called logistic regression, and suggests how AdaBoost's predictions can be used to estimate the probability of a particular example being positive or negative.

From yet another vantage point, which turns out to be “dual” to the one given in chapter 7, AdaBoost can be interpreted in a kind of abstract, geometric framework. Here, the fundamental operation is projection of a point onto a subspace. In this case, the “points” are

in fact the distributions D_t computed by AdaBoost, which exist in a kind of “information geometric” space—one based on notions from information theory—rather than the usual Euclidean geometry. As discussed in chapter 8, this view leads to a deeper understanding of AdaBoost’s dynamics and underlying mathematical structure, and yields proofs of fundamental convergence properties.

Part III of this book focuses on practical, algorithmic extensions of AdaBoost. In the basic form shown in algorithm 1.1, AdaBoost is intended for the simplest learning setting in which the goal is binary classification, that is, classification problems with only two possible classes or categories. To apply AdaBoost to a much broader range of real-world learning problems, the algorithm must be extended along multiple dimensions.

In chapter 9, we describe an extension to AdaBoost in which the base classifiers themselves are permitted to output predictions that vary in their self-rated level of confidence. In practical terms, this modification of boosting leads to a dramatic speedup in learning time. Moreover, within this framework we derive two algorithms designed to produce classifiers that are not only accurate, but also understandable in form to humans.

Chapter 10 extends AdaBoost to the case in which there are more than two possible classes, as is very commonly the case in actual applications. For instance, if recognizing digits, there are ten classes, one for each digit. As will be seen, it turns out that there are quite a number of methods for modifying AdaBoost for this purpose, and we will see how a great many of these can be studied in a unified framework.

Chapter 11 extends AdaBoost to ranking problems, that is, problems in which the goal is to learn to rank a set of objects. For instance, the goal might be to rank credit card transactions according to the likelihood of each one being fraudulent, so that those at the top of the ranking can be investigated.

Finally, in part IV, we study a number of advanced theoretical topics.

The first of these provides an alternative approach for the understanding of AdaBoost’s generalization capabilities, which explicitly takes into consideration intrinsic randomness or “noise” in the data that may prevent perfect generalization by *any* classifier. In such a setting, we show in chapter 12 that the accuracy of AdaBoost will nevertheless converge to that of the best possible classifier, under appropriate assumptions. However, we also show that without these assumptions, AdaBoost’s performance can be rather poor when the data is noisy.

AdaBoost can be understood in many ways, but at its foundation, it is a boosting algorithm in the original technical meaning of the word, a provable method for driving down the error of the combined classifier by combining a number of weak classifiers. In fact, for this specific problem, AdaBoost is not the best possible; rather, there is another algorithm called “boost-by-majority” that is optimal in a very strong sense, as we will see in chapter 13. However, this latter algorithm is not practical because it is not adaptive in the sense described in section 1.2.1. Nevertheless, as we show in chapter 14, this algorithm can be made adaptive by taking a kind of limit in which the discrete time steps in the usual boosting framework are replaced by a *continuous* sequence of time steps. This leads to the “BrownBoost”

algorithm, which has certain properties that suggest greater tolerance to noise, and from which AdaBoost can be derived in the “zero-noise” limit.

Although this book is about foundations and algorithms, we also provide numerous examples illustrating how the theory we develop can be applied practically. Indeed, as seen earlier in this chapter, AdaBoost has many practical advantages. It is fast, simple, and easy to program. It has no parameters to tune (except for the number of rounds T). It requires no prior knowledge about the base learner, and so can be flexibly combined with any method for finding base classifiers. Finally, it comes with a set of theoretical guarantees, given sufficient data and a base learner that can reliably provide only moderately accurate base classifiers. This is a shift in mind-set for the learning-system designer: instead of trying to design a learning algorithm that is accurate over the entire space, we can instead focus on finding weak learning algorithms that only need to be better than random.

On the other hand, some caveats are certainly in order. The actual performance of boosting on a particular problem is clearly dependent on the data and the base learner. Consistent with the theory outlined above and discussed in detail in this book, boosting can fail to perform well, given insufficient data, overly complex base classifiers, or base classifiers that are too weak. Boosting seems to be especially susceptible to noise, as we discuss in section 12.3. Nonetheless, as seen in section 1.2.2, on a wide range of real-world learning problems, boosting’s performance overall is quite good.

To illustrate its empirical performance and application, throughout this book we give examples of its use on practical problems such as human-face detection, topic identification, language understanding in spoken-dialogue systems, and natural-language parsing.

Summary

In this chapter, we have given an introduction to machine learning, classification problems, and boosting, particularly AdaBoost and its variants, which are the focus of this book. We have presented examples of boosting’s empirical performance, as well as an overview of some of the highlights of its rich and varied theory. In the chapters ahead, we explore the foundations of boosting from many vantage points, and develop key principles in the design of boosting algorithms, while also giving examples of their application to practical problems.

Bibliographic Notes

Boosting has its roots in a theoretical framework for studying machine learning called the PAC model, proposed by Valiant [221], which we discuss in more detail in section 2.3. Working in this framework, Kearns and Valiant [133] posed the question of whether a weak learning algorithm that performs just slightly better than random guessing can be boosted into one with arbitrarily high accuracy. Schapire [199] came up with the first provable polynomial-time boosting algorithm in 1989. A year later, Freund [88] developed a much

more efficient boosting algorithm called boost-by-majority that is essentially optimal (see chapter 13). The first experiments with these early boosting algorithms were carried out by Drucker, Schapire, and Simard [72] on an OCR task. However, both algorithms were largely impractical because of their nonadaptiveness. AdaBoost, the first adaptive boosting algorithm, was introduced in 1995 by Freund and Schapire [95].

There are many fine textbooks which provide a broader treatment of machine learning, a field that overlaps considerably with statistics, pattern recognition, and data mining. See, for instance, [7, 22, 67, 73, 120, 134, 166, 171, 223, 224]. For alternative surveys of boosting and related methods for combining classifiers, see refs. [40, 69, 146, 170, 214].

The medical-diagnosis data used in section 1.2.3 was collected from the Cleveland Clinic Foundation by Detrano et al. [66]. The letter recognition dataset used in section 1.3 was created by Frey and Slate [97]. The C4.5 decision-tree learning algorithm used in sections 1.2.2 and 1.3 is due to Quinlan [184], and is similar to the CART algorithm of Breiman et al. [39].

Drucker and Cortes [71] and Jackson and Craven [126] were the first to test AdaBoost experimentally. The experiments in section 1.2.2 were originally reported by Freund and Schapire [93] from which the right plot of figure 1.3 and left plot of figure 1.4 were adapted. AdaBoost's resistance to overfitting was noticed early on by Drucker and Cortes [71], as well as by Breiman [35] and Quinlan [183]. The experiments in section 1.3, including figure 1.7, are taken from Schapire et al. [202]. There have been numerous other systematic experimental studies of AdaBoost, such as [15, 68, 162, 209], as well as Caruana and Niculescu-Mizil's [42] large-scale comparison of several learning algorithms, including AdaBoost.

Exercises

1.1 Show that the error of h_t on distribution D_{t+1} is exactly $\frac{1}{2}$, that is,

$$\Pr_{i \sim D_{t+1}}[h_t(x_i) \neq y_i] = \frac{1}{2}.$$

1.2 For each of the following cases, explain how AdaBoost, as given in algorithm 1.1, will treat a weak hypothesis h_t with weighted error ϵ_t . Also, in each case, explain how this behavior makes sense.

- a. $\epsilon_t = \frac{1}{2}$.
- b. $\epsilon_t > \frac{1}{2}$.
- c. $\epsilon_t = 0$.

1.3 In figure 1.7, the training error and test error of the combined classifier H are seen to increase significantly on the second round. Give a plausible explanation why we might expect these error rates to be higher after two rounds than after only one.