Chapter 12

Large-Scale Machine Learning

Many algorithms are today classified as "machine learning." These algorithms share, with the other algorithms studied in this book, the goal of extracting information from data. All algorithms for analysis of data are designed to produce a useful summary of the data, from which decisions are made. Among many examples, the frequent-itemset analysis that we did in Chapter 6 produces information like association rules, which can then be used for planning a sales strategy or for many other purposes.

However, algorithms called "machine learning" not only summarize our data; they are perceived as learning a model or classifier from the data, and thus discover something about data that will be seen in the future. For instance, the clustering algorithms discussed in Chapter 7 produce clusters that not only tell us something about the data being analyzed (the training set), but they allow us to classify future data into one of the clusters that result from the clustering algorithm. Thus, machine-learning enthusiasts often speak of clustering with the neologism "unsupervised learning"; the term *unsupervised* refers to the fact that the input data does not tell the clustering algorithm what the clusters should be. In *supervised* machine learning,, which is the subject of this chapter, the available data includes information about the correct way to classify at least some of the data. The data classified already is called the *training set*.

This chapter is not intended to be a complete discussion of machine learning. We concentrate on a small number of ideas, and emphasize how to deal with very large data sets. Especially important is how we exploit parallelism to build models of the data. We consider the classical "perceptron" approach to learning a data classifier, where a hyperplane that separates two classes is sought. Then, we look at more modern techniques involving support-vector machines. Similar to perceptrons, these methods look for hyperplanes that best divide the classes, so that few, if any, members of the training set lie close to the hyperplane. We next consider nearest-neighbor techniques, where data is classified according to

the class(es) of their nearest neighbors in some space. We end with a discussion of decision trees, which are branching programs for predicting the class of an example.

12.1 The Machine-Learning Model

In this section we introduce the framework for machine-learning algorithms and give the basic definitions.

12.1.1 Training Sets

The data to which a machine-learning (often abbreviated ML) algorithm is applied is called a training set. A *training set* consists of a set of pairs $(\mathbf{x}, y)$, called *training examples*, where

- $\mathbf{x}$ is a vector of values, often called a *feature vector*, or simply an *input*. Each value, or feature, can be *categorical* (values are taken from a set of discrete values, such as {red, blue, green}) or *numerical* (values are integers or real numbers).
- y is the *class label*, or simply *output*, the classification value for $\mathbf{x}$.

The objective of the ML process is to discover a function $y = f(\mathbf{x})$ that best predicts the value of y associated with each value of $\mathbf{x}$. The type of y is in principle arbitrary, but there are several common and important cases.

1. y is a real number. In this case, the ML problem is called *regression*.
2. y is a Boolean value: true-or-false, more commonly written as $+1$ and -1, respectively. In this class the problem is *binary classification*.
3. y is a member of some finite set. The members of this set can be thought of as "classes," and each member represents one class. The problem is *multiclass classification*.
4. y is a member of some potentially infinite set, for example, a parse tree for x, which is interpreted as a sentence.

12.1.2 Some Illustrative Examples

Example 12.1 : Recall Fig. 7.1, repeated as Fig. 12.1, where we plotted the height and weight of dogs in three classes: Beagles, Chihuahuas, and Dachshunds. We can think of this data as a training set, provided the data includes the variety of the dog along with each height-weight pair. Each pair $(\mathbf{x}, y)$ in the training set consists of a feature vector $\mathbf{x}$ of the form [height, weight]. The associated label y is the variety of the dog. An example of a training-set pair would be ([5 inches, 2 pounds], Chihuahua).

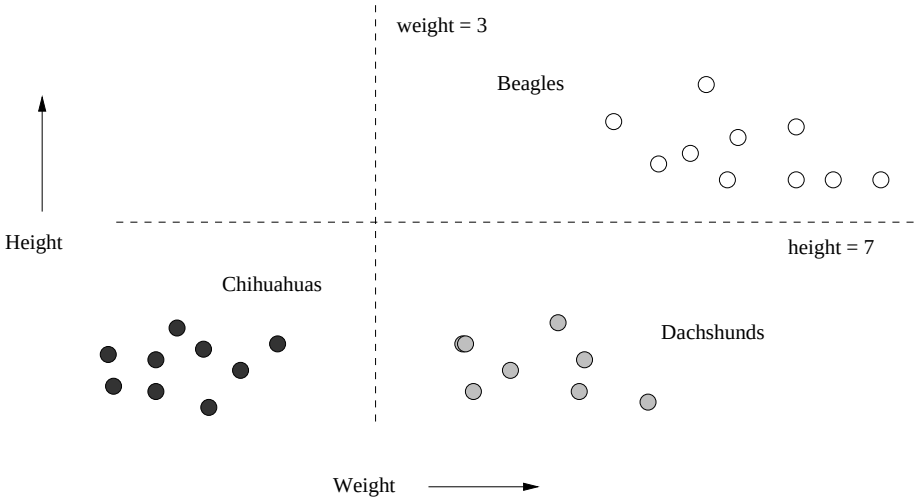


Figure 12.1: Repeat of Fig. 7.1, indicating the heights and weights of certain dogs

An appropriate way to implement the decision function f would be to imagine two lines, shown dashed in Fig. 12.1. The horizontal line represents a height of 7 inches and separates Beagles from Chihuahuas and Dachshunds. The vertical line represents a weight of 3 pounds and separates Chihuahuas from Beagles and Dachshunds. The algorithm that implements f is:

```
if (height > 7) print Beagle
else if (weight < 3) print Chihuahua
else print Dachshund;
```

Recall that the original intent of Fig. 7.1 was to cluster points without knowing which variety of dog they represented. That is, the label associated with a given height-weight vector was not available. Here, we are performing supervised learning with the same data augmented by classifications for the training data. □

Example 12.2: As an example of supervised learning, the four points $(1, 2)$, $(2, 1)$, $(3, 4)$, and $(4, 3)$ from Fig.11.1 (repeated here as Fig. 12.2), can be thought of as a training set, where the vectors are one-dimensional. That is, the point $(1, 2)$ can be thought of as a pair $([1], 2)$, where $[1]$ is the one-dimensional feature vector $\mathbf{x}$, and 2 is the associated label y; the other points can be interpreted similarly.

Suppose we want to "learn" the best model for the full set of points, from which these four points are taken as examples. A simple form of model, the one we shall consider, is a linear function $f(x) = ax + b$. A natural interpretation

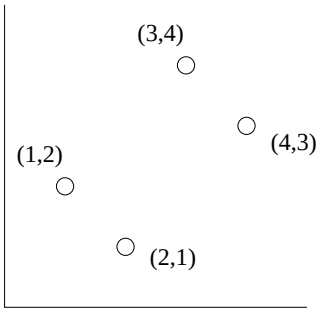


Figure 12.2: Repeat of Fig. 11.1, to be used as a training set

of "best" is that the RMSE of the value of $f(x)$ compared with the given value of y is minimized. That is, we want to minimize

$$\sum_{x=1}^{4}(ax + b - y_x)^2$$

where y_x is the y-value associated with x. This sum is

$$(a + b - 2)^2 + (2a + b - 1)^2 + (3a + b - 4)^2 + (4a + b - 3)^2$$

Simplifying, the sum is $30a^2 + 4b^2 + 20ab - 56a - 20b + 30$. If we then take the derivatives with respect to a and b and set them to 0, we get

$$\begin{aligned} 60a + 20b - 56 &= 0 \\ 20a + 8b - 20 &= 0 \end{aligned}$$

The solution to these equations is $a = 3/5$ and $b = 1$. For these values the RMSE is 3.2.

Note that the learned straight line is not the principal axis that was discovered for these points in Section 11.2.1. That axis was the line with slope 1, going through the origin, i.e., the line $y = x$. For this line, the RMSE is 4. The difference is that PCA discussed in Section 11.2.1 minimizes the sum of the squares of the lengths of the projections onto the chosen axis, which is constrained to go through the origin. Here, we are minimizing the sum of the squares of the vertical distances between the points and the line. In fact, even had we tried to learn the line through the origin with the least RMSE, we would not choose the line $y = x$. You can check that $y = \frac{14}{15}x$ has a lower RMSE than 4. □

Example 12.3: A common application of machine learning involves a training set where the feature vectors $\mathbf{x}$ are Boolean-valued and of very high dimension. We shall focus on data consisting of documents, e.g., emails, Web pages, or newspaper articles. Each component represents a word in some large dictionary. We would probably eliminate stop words (very common words) from this

dictionary, because these words tend not to tell us much about the subject matter of the document. Similarly, we might also restrict the dictionary to words with high TF.IDF scores (see Section 1.3.1) so the words considered would tend to reflect the topic or substance of the document.

The training set consists of pairs, where the vector $\mathbf{x}$ represents the presence or absence of each dictionary word in document. The label y could be $+1$ or -1, with $+1$ representing that the document (an email, e.g.) is spam. Our goal would be to train a classifier to examine future emails and decide whether or not they are spam. We shall illustrate this use of machine learning in Example 12.4.

Alternatively, y could be chosen from some finite set of topics, e.g., "sports," "politics," and so on. Again, $\mathbf{x}$ could represent a document, perhaps a Web page. The goal would be to create a classifier for Web pages that assigned a topic to each. □

12.1.3 Approaches to Machine Learning

There are many forms of ML algorithms, and we shall not cover them all here. Here are the major classes of such algorithms, each of which is distinguished by the form by which the function f is represented.

1. *Decision trees* were discussed briefly in Section 9.2.7 and will be covered more extensively in Section 12.5. The form of f is a tree, and each node of the tree has a function of $\mathbf{x}$ that determines to which child or children the search must proceed. Decision trees are suitable for binary and multiclass classification, especially when the dimension of the feature vector is not too large (large numbers of features can lead to overfitting).

2. *Perceptrons* are threshold functions applied to the components of the vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$. A weight w_i is associated with the ith component, for each $i = 1, 2, \ldots, n$, and there is a threshold θ. The output is $+1$ if

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

   and the output is -1 if that sum is less than θ. A perceptron is suitable for binary classification, even when the number of features is very large, e.g., the presence or absence of words in a document. Perceptrons are the topic of Section 12.2.

3. *Neural nets* are acyclic networks of perceptrons, with the outputs of some perceptrons used as inputs to others. These are suitable for binary or multiclass classification, since there can be several perceptrons used as output, with one or more indicating each class.

4. *Instance-based learning* uses the entire training set to represent the function f. The calculation of the label y associated with a new feature vector $\mathbf{x}$ can involve examination of the entire training set, although usually some

preprocessing of the training set enables the computation of $f(\mathbf{x})$ to proceed efficiently. We shall consider an important kind of instance-based learning, k-nearest-neighbor, in Section 12.4. For example, 1-nearest-neighbor classifies data by giving it the same class as that of its nearest training example. There are k-nearest-neighbor algorithms that are appropriate for any kind of classification, although we shall concentrate on the case where y and the components of $\mathbf{x}$ are real numbers.

5. *Support-vector machines* are an advance over the algorithms traditionally used to select the weights and threshold. The result is a classifier that tends to be more accurate on unseen data. We discuss support-vector machines in Section 12.3.

12.1.4 Machine-Learning Architecture

Machine-learning algorithms can be classified not only by their general algorithmic approach as we discussed in Section 12.1.3. but also by their underlying architecture – the way data is handled and the way it is used to build the model.

Training, Validating, and Testing

One general issue regarding the handling of data is that there is a good reason to withhold some of the available data from the training set. The remaining data is called the *test set*. In some cases, we withhold two sets of training examples, a *validation set* as well as the test set. The difference is that the validation set is used to help design the model, while the test set is used only to determine how good the model is. The problem addressed by a validation set is that many machine-learning algorithms tend to *overfit* the data; they pick up on artifacts that occur in the training set but that are atypical of the larger population of possible data. By using the vaidation set, and seeing how well the classifier works on that, we can tell if the classifier is overfitting the data. If so, we can restrict the machine-learning algorithm in some way. For instance, if we are constructing a decision tree, we can limit the number of levels of the tree.

Figure 12.3 illustrates the train-and-test architecture. We assume all the data is suitable for training (i.e., the class information is attached to the data), but we separate out a small fraction of the available data as the test set. We use the remaining data to build a suitable model or classifier. Then we feed the test data to this model. Since we know the class of each element of the test data, we can tell how well the model does on the test data. If the error rate on the test data is not much worse than the error rate of the model on the training data itself, then we expect there is little, if any, overfitting, and the model can be used. On the other hand, if the classifier performs much worse on the test data than on the training data, we expect there is overfitting and need to rethink the way we construct the classifier.

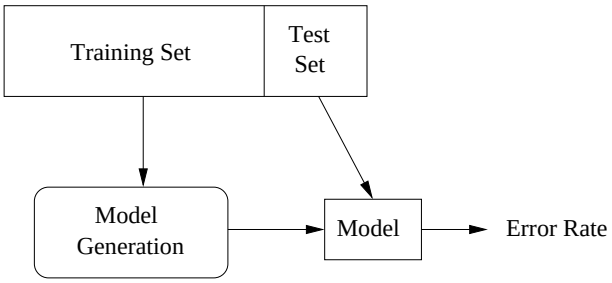


Figure 12.3: The training set helps build the model, and the test set validates it

Generalization

We should remember that the purpose of creating a model or classifier is not to classify the training set, but to classify the data whose class we do not know. We want that data to be classified correctly, but often we have no way of knowing whether or not the model does so. If the nature of the data changes over time, for instance, if we are trying to detect spam emails, then we need to measure the performance over time, as best we can. For example, in the case of spam emails, we can note the rate of reports of spam emails that were not classified as spam.

There is nothing special about the selection of the test data. In fact, we can repeat the train-then-test process several times using the same data, if we divide the data into k equal-sized chunks. In turn, we let each chunk be the test data, and use the remaining $k-1$ chunks as the training data. This training architecture is called *cross-validation*.

Batch Versus On-Line Learning

Often, as in Examples 12.1 and 12.2, we use a *batch learning* architecture. That is, the entire training set is available at the beginning of the process, and it is all used in whatever way the algorithm requires to produce a model once and for all. The alternative is *on-line learning*, where the training set arrives in a stream and, like any stream, cannot be revisited after it is processed. In on-line learning, we maintain a model at all times. As new training examples arrive, we may choose to modify the model to account for the new examples. On-line learning has the advantages that it can

1. Deal with very large training sets, because it does not access more than one training example at a time.

2. Adapt to changes in the population of training examples as time goes on. For instance, Google trains its spam-email classifier this way, adapting the classifier for spam as new kinds of spam email are sent by spammers and indicated to be spam by the recipients.

An enhancement of on-line learning, suitable in some cases, is *active learning*. Here, the classifier may receive some training examples, but it primarily receives unclassified data, which it must classify. If the classifier is unsure of the classification (e.g., the newly arrived example is very close to the boundary), then the classifier can ask for ground truth at some significant cost. For instance, it could send the example to Mechanical Turk and gather opinions of real people. In this way, examples near the boundary become training examples and can be used to modify the classifier.

Feature Selection

Sometimes, the hardest part of designing a good model or classifier is figuring out what features to use as input to the learning algorithm. Let us reconsider Example 12.3, where we suggested that we could classify emails as spam or not spam by looking at the words contained in the email. In fact, we explore in detail such a classifier in Example 12.4. As discussed in Example 12.3, it may make sense to focus on certain words and not others; e.g., we should eliminate stop words.

But we should also ask whether there is other information available that would help us make a better decision about spam. For example, spam is often generated by particular hosts, either those belonging to the spammers, or hosts that have been coopted into a "botnet" for the purpose of generating spam. Thus, including the originating host or originating email address into the feature vector describing an email might enable us to design a better classifier and lower the error rate.

Creating a Training Set

It is reasonable to ask where the label information that turns data into a training set comes from. The obvious method is to create the labels by hand, having an expert look at each feature vector and classify it properly. Recently, crowdsourcing techniques have been used to label data. For example, in many applications it is possible to use Mechanical Turk to label data. Since the "Turkers" are not necessarily reliable, it is wise to use a system that allows the question to be asked of several different people, until a clear majority is in favor of one label.

One often can find data on the Web that is implicitly labeled. For example, the Open Directory (DMOZ) has millions of pages labeled by topic. That data, used as a training set, can enable one to classify other pages or documents according to their topic, based on the frequency of word occurrence. Another approach to classifying by topic is to look at the Wikipedia page for a topic and

see what pages it links to. Those pages can safely be assumed to be relevant to the given topic.

In some applications we can use the stars that people use to rate products or services on sites like Amazon or Yelp. For example, we might want to estimate the number of stars that would be assigned to reviews or tweets about a product, even if those reviews or tweets do not themselves have star ratings. If we use star-labeled reviews as a training set, we can deduce the words that are most commonly associated with positive and negative reviews (called *sentiment analysis*). The presence of these words in other reviews can tell us the sentiment of those reviews.

12.1.5 Exercises for Section 12.1

Exercise 12.1.1: Redo Example 12.2 for the following different forms of $f(x)$.

(a) Require $f(x) = ax$; i.e., a straight line through the origin. Is the line $y = \frac{14}{15}x$ that we discussed in the example optimal?

(b) Allow $f(x)$ to be a quadratic, i.e., $f(x) = ax^2 + bx + c$.

12.2 Perceptrons

A *perceptron* is a linear binary classifier. Its input is a vector $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ with real-valued components. Associated with the perceptron is a vector of *weights* $\mathbf{w} = [w_1, w_2, \ldots, w_d]$, also with real-valued components. Each perceptron has a *threshold* θ. The output of the perceptron is $+1$ if $\mathbf{w}.\mathbf{x} > \theta$, and the output is -1 if $\mathbf{w}.\mathbf{x} < \theta$. The special case where $\mathbf{w}.\mathbf{x} = \theta$ will always be regarded as "wrong," in the sense that we shall describe in detail when we get to Section 12.2.1.

The weight vector $\mathbf{w}$ defines a hyperplane of dimension $d - 1$ – the set of all points $\mathbf{x}$ such that $\mathbf{w}.\mathbf{x} = \theta$, as suggested in Fig. 12.4. Points on the positive side of the hyperplane are classified $+1$ and those on the negative side are classified -1. A perceptron classifier works only for data that is *linearly separable*, in the sense that there is some hyperplane that separates all the positive points from all the negative points. If there are many such hyperplanes, the perceptron will converge to one of them, and thus will correctly classify all the training points. If no such hyperplane exists, then the perceptron cannot converge to any hyperplane. In the next section, we discuss support-vector machines, which do not have this limitation; they will converge to some separator that, although not a perfect classifier, will do as well as possible under the metric to be described in Section 12.3.

12.2.1 Training a Perceptron with Zero Threshold

To train a perceptron, we examine the training set and try to find a weight vector $\mathbf{w}$ and threshold θ such that all the feature vectors with $y = +1$ (the

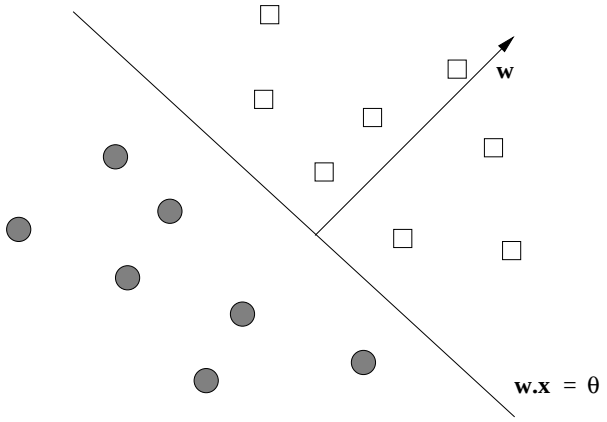


Figure 12.4: A perceptron divides a space by a hyperplane into two half-spaces

positive examples) are on the positive side of the hyperplane and all those with $y = -1$ (the *negative examples*) are on the negative side. It may or may not be possible to do so, since there is no guarantee that *any* hyperplane separates all the positive and negative examples in the training set.

We begin by assuming the threshold is 0; the simple augmentation needed to handle an unknown threshold is discussed in Section 12.2.4. The following method will converge to some hyperplane that separates the positive and negative examples, provided one exists.

1. Initialize the weight vector $\mathbf{w}$ to all 0's.
2. Pick a *learning-rate parameter* η, which is a small, positive real number. The choice of η affects the convergence of the perceptron. If η is too small, then convergence is slow; if it is too big, then the decision boundary will "dance around" and again will converge slowly, if at all.
3. Consider each training example $t = (\mathbf{x}, y)$ in turn.
   (a) Let $y' = \mathbf{w}.\mathbf{x}$.
   (b) If y' and y have the same sign, then do nothing; t is properly classified.
   (c) However, if y' and y have different signs, or $y' = 0$, replace $\mathbf{w}$ by $\mathbf{w} + \eta y \mathbf{x}$. That is, adjust $\mathbf{w}$ slightly in the direction of $\mathbf{x}$.

The two-dimensional case of this transformation on $\mathbf{w}$ is suggested in Fig. 12.5. Notice how moving $\mathbf{w}$ in the direction of $\mathbf{x}$ moves the hyperplane that is perpendicular to $\mathbf{w}$ in such a direction that it makes it more likely that $\mathbf{x}$ will be on the correct side of the hyperplane, although it does not guarantee that $\mathbf{x}$ will then be correctly classified.

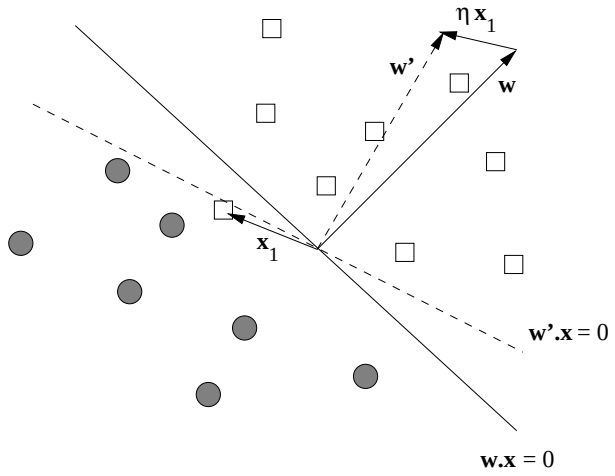


Figure 12.5: A misclassified point $\mathbf{x}_1$ moves the vector $\mathbf{w}$

Example 12.4: Let us consider training a perceptron to recognize spam email. The training set consists of pairs $(\mathbf{x}, y)$ where $\mathbf{x}$ is a vector of 0's and 1's, with each component x_i corresponding to the presence ($x_i = 1$) or absence ($x_i = 0$) of a particular word in the email. The value of y is $+1$ if the email is known to be spam and -1 if it is known not to be spam. While the number of words found in the training set of emails is very large, we shall use a simplified example where there are only five words: "and," "viagra," "the," "of," and "nigeria." Figure 12.6 gives the training set of six vectors and their corresponding classes.

	and	viagra	the	of	nigeria	y
a	1	1	0	1	1	+1
b	0	0	1	1	0	−1
c	0	1	1	0	0	+1
d	1	0	0	1	0	−1
e	1	0	1	0	1	+1
f	1	0	1	1	0	−1

Figure 12.6: Training data for spam emails

In this example, we shall use learning rate $\eta = 1/2$, and we shall visit each training example once, in the order shown in Fig. 12.6. We begin with $\mathbf{w} = [0, 0, 0, 0, 0]$ and compute $\mathbf{w}.\mathbf{a} = 0$. Since 0 is not positive, we move $\mathbf{w}$ in the direction of $\mathbf{a}$ by performing $\mathbf{w} := \mathbf{w} + (1/2)(+1)\mathbf{a}$. The new value of $\mathbf{w}$ is thus

$$\mathbf{w} = [0, 0, 0, 0, 0] + [\frac{1}{2}, \frac{1}{2}, 0, \frac{1}{2}, \frac{1}{2}] = [\frac{1}{2}, \frac{1}{2}, 0, \frac{1}{2}, \frac{1}{2}]$$

Next, consider $\mathbf{b}$. $\mathbf{w}.\mathbf{b} = [\frac{1}{2}, \frac{1}{2}, 0, \frac{1}{2}, \frac{1}{2}].[0, 0, 1, 1, 0] = \frac{1}{2}$. Since the associated

Pragmatics of Training on Emails

When we represent emails or other large documents as training examples, we would not really want to construct the vector of 0's and 1's with a component for every word that appears even once in the collection of emails. Doing so would typically give us sparse vectors with millions of components. Rather, create a table in which all the words appearing in the emails are assigned integers $1, 2, \ldots$, indicating their component. When we process an email in the training set, make a list of the components in which the vector has 1; i.e., use the standard sparse representation for the vector. If we eliminate stop words from the representation, or even eliminate words with a low TF.IDF score, then we make the vectors representing emails significantly sparser and thus compress the data even more. Only the vector $\mathbf{w}$ needs to have all its components listed, since it will not be sparse after a small number of training examples have been processed.

y for $\mathbf{b}$ is -1, $\mathbf{b}$ is misclassified. We thus assign

$$\mathbf{w} := \mathbf{w} + (1/2)(-1)\mathbf{b} = [\frac{1}{2}, \frac{1}{2}, 0, \frac{1}{2}, \frac{1}{2}] - [0, 0, \frac{1}{2}, \frac{1}{2}, 0] = [\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, 0, \frac{1}{2}]$$

Training example $\mathbf{c}$ is next. We compute

$$\mathbf{w}.\mathbf{c} = [\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, 0, \frac{1}{2}].[0, 1, 1, 0, 0] = 0$$

Since the associated y for $\mathbf{c}$ is $+1$, $\mathbf{c}$ is also misclassified. We thus assign

$$\mathbf{w} := \mathbf{w} + (1/2)(+1)\mathbf{c} = [\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}, 0, \frac{1}{2}] + [0, \frac{1}{2}, \frac{1}{2}, 0, 0] = [\frac{1}{2}, 1, 0, 0, \frac{1}{2}]$$

Training example $\mathbf{d}$ is next to be considered:

$$\mathbf{w}.\mathbf{d} = [\frac{1}{2}, 1, 0, 0, \frac{1}{2}].[1, 0, 0, 1, 0] = 1$$

Since the associated y for $\mathbf{d}$ is -1, $\mathbf{d}$ is misclassified as well. We thus assign

$$\mathbf{w} := \mathbf{w} + (1/2)(-1)\mathbf{d} = [\frac{1}{2}, 1, 0, 0, \frac{1}{2}] - [\frac{1}{2}, 0, 0, \frac{1}{2}, 0] = [0, 1, 0, -\frac{1}{2}, \frac{1}{2}]$$

For training example $\mathbf{e}$ we compute $\mathbf{w}.\mathbf{e} = [0, 1, 0, -\frac{1}{2}, \frac{1}{2}].[1, 0, 1, 0, 1] = \frac{1}{2}$. Since the associated y for $\mathbf{e}$ is $+1$, $\mathbf{e}$ is classified correctly, and no change to $\mathbf{w}$ is made. Similarly, for $\mathbf{f}$ we compute

$$\mathbf{w}.\mathbf{f} = [0, 1, 0, -\frac{1}{2}, \frac{1}{2}].[1, 0, 1, 1, 0] = -\frac{1}{2}$$

so **f** is correctly classified. If we check **a** through **d**, we find that this **w** correctly classifies them as well. Thus, we have converged to a perceptron that classifies all the training set examples correctly. It also makes a certain amount of sense: it says that "viagra" and "nigeria" are indicative of spam, while "of" is indicative of nonspam. It considers "and" and "the" neutral," although we would probably prefer to give "and," "of," and "the" the same weight. □

12.2.2 Convergence of Perceptrons

As we mentioned at the beginning of this section, if the data points are linearly separable, then the perceptron algorithm will converge to a separator. However, if the data is not linearly separable, then the algorithm will eventually repeat a weight vector and loop infinitely. Unfortunately, it is often hard to tell, during the running of the algorithm, which of these two cases applies. When the data is large, it is not feasible to remember all previous weight vectors to see whether we are repeating a vector, and even if we could, the period of repetition would most likely be so large that we would want to terminate the algorithm long before we repeated.

A second issue regarding termination is that even if the training data is linearly separable, the entire dataset might not be linearly separable. The consequence is that there might not be any value in running the algorithm for a very large number of rounds, in the hope of converging to a separator. We therefore need a strategy for deciding when to terminate the perceptron algorithm, assuming convergence has not occurred. Here are some common tests for termination.

1. Terminate after a fixed number of rounds.
2. Terminate when the number of misclassified training points stops changing.
3. Withhold a test set from the training data, and after each round, run the perceptron on the test data. Terminate the algorithm when the number of errors on the test set stops changing.

Another technique that will aid convergence is to lower the training rate as the number of rounds increases. For example, we could allow the training rate η to start at some initial η_0 and lower it to $\eta_0/(1 + ct)$ after the tth round, where c is some small constant.

12.2.3 The Winnow Algorithm

There are many other rules one could use to adjust weights for a perceptron. Not all possible algorithms are guaranteed to converge, even if there is a hyperplane separating positive and negative examples. One that does converge is called *Winnow*, and that rule will be described here. Winnow assumes that the feature

vectors consist of 0's and 1's, and the labels are $+1$ or -1. Unlike the basic perceptron algorithm, which can produce positive or negative components in the weight vector $\mathbf{w}$, Winnow produces only positive weights.

The general Winnow Algorithm allows for a variety of parameters to be selected, and we shall only consider one simple variant. However, all variants have in common the idea that there is a positive threshold θ. If $\mathbf{w}$ is the current weight vector, and $\mathbf{x}$ is the feature vector in the training set that we are currently considering, we compute $\mathbf{w}.\mathbf{x}$ and compare it with θ. If $\mathbf{w}.\mathbf{x} \leq \theta$, and the class for $\mathbf{x}$ is $+1$, then we have to raise the weights of $\mathbf{w}$ in those components where $\mathbf{x}$ has 1. We multiply these weights by a number greater than 1. The larger this number, the greater the training rate, so we want to pick a number that is not too close to 1 (or convergence will be too slow) but also not too large (or the weight vector may oscillate). Similarly, if $\mathbf{w}.\mathbf{x} \geq \theta$, but the class of $\mathbf{x}$ is -1, then we want to lower the weights of $\mathbf{w}$ in those components where $\mathbf{x}$ is 1. We multiply those weights by a number greater than 0 but less than 1. Again, we want to pick a number that is not too close to 1 but also not too small, to avoid slow convergence or oscillation.

We shall give the details of the algorithm using the factors 2 and 1/2, for the cases where we want to raise weights and lower weights, respectively. Start the Winnow Algorithm with a weight vector $\mathbf{w} = [w_1, w_2, \ldots, w_d]$ all of whose components are 1, and let the threshold θ equal d, the number of dimensions of the vectors in the training examples. Let $(\mathbf{x}, y)$ be the next training example to be considered, where $\mathbf{x} = [x_1, x_2, \ldots, x_d]$.

1. If $\mathbf{w}.\mathbf{x} > \theta$ and $y = +1$, or $\mathbf{w}.\mathbf{x} < \theta$ and $y = -1$, then the example is correctly classified, so no change to $\mathbf{w}$ is made.

2. If $\mathbf{w}.\mathbf{x} \leq \theta$, but $y = +1$, then the weights for the components where $\mathbf{x}$ has 1 are too low as a group. Double each of the corresponding components of $\mathbf{w}$. That is, if $x_i = 1$ then set $w_i := 2w_i$.

3. If $\mathbf{w}.\mathbf{x} \geq \theta$, but $y = -1$, then the weights for the components where $\mathbf{x}$ has 1 are too high as a group. Halve each of the corresponding components of $\mathbf{w}$. That is, if $x_i = 1$ then set $w_i := w_i/2$.

Example 12.5: Let us reconsider the training data from Fig. 12.6. Initialize $\mathbf{w} = [1, 1, 1, 1, 1]$ and let $\theta = 5$. First, consider feature vector $\mathbf{a} = [1, 1, 0, 1, 1]$. $\mathbf{w}.\mathbf{a} = 4$, which is less than θ. Since the associated label for $\mathbf{a}$ is $+1$, this example is misclassified. When a $+1$-labeled example is misclassified, we must double all the components where the example has 1; in this case, all but the third component of $\mathbf{a}$ is 1. Thus, the new value of $\mathbf{w}$ is $[2, 2, 1, 2, 2]$.

Next, we consider training example $\mathbf{b} = [0, 0, 1, 1, 0]$. $\mathbf{w}.\mathbf{b} = 3$, which is less than θ. However, the associated label for $\mathbf{b}$ is -1, so no change to $\mathbf{w}$ is needed.

For $\mathbf{c} = [0, 1, 1, 0, 0]$ we find $\mathbf{w}.\mathbf{c} = 3 < \theta$, while the associated label is $+1$. Thus, we double the components of $\mathbf{w}$ where the corresponding components of $\mathbf{c}$ are 1. These components are the second and third, so the new value of $\mathbf{w}$ is $[2, 4, 2, 2, 2]$.

The next two training examples, **d** and **e** require no change, since they are correctly classified. However, there is a problem with $\mathbf{f} = [1, 0, 1, 1, 0]$, since $\mathbf{w}.\mathbf{f} = 6 > \theta$, while the associated label for **f** is -1. Thus, we must divide the first, third, and fourth components of **w** by 2, since these are the components where **f** has 1. The new value of **w** is $[1, 4, 1, 1, 2]$.

x	y	**w.x**	OK?	and	viagra	the	of	nigeria
				1	1	1	1	1
a	+1	4	no	2	2	1	2	2
b	−1	3	yes					
c	+1	3	no	2	4	2	2	2
d	−1	4	yes					
e	+1	6	yes					
f	−1	6	no	1	4	1	1	2
a	+1	8	yes					
b	−1	2	yes					
c	+1	5	no	1	8	2	1	2
d	−1	2	yes					
e	+1	5	no	2	8	4	1	4
f	−1	7	no	1	8	2	$\frac{1}{2}$	4

Figure 12.7: Sequence of updates to **w** performed by the Winnow Algorithm on the training set of Fig. 12.6

We still have not converged. It turns out we must consider each of the training examples **a** through **f** again. At the end of this process, the algorithm has converged to a weight vector $\mathbf{w} = [1, 8, 2, \frac{1}{2}, 4]$, which with threshold $\theta = 5$ correctly classifies all of the training examples in Fig. 12.6. The details of the twelve steps to convergence are shown in Fig. 12.7. This figure gives the associated label y and the computed dot product of **w** and the given feature vector. The last five columns are the five components of **w** after processing each training example. □

12.2.4 Allowing the Threshold to Vary

Suppose now that the choice of threshold 0, as in Section 12.2.1, or threshold d, as in Section 12.2.3 is not desirable, or that we don't know the best threshold to use. At the cost of adding another dimension to the feature vectors, we can treat θ as one of the components of the weight vector **w**. That is:

1. Replace the vector of weights $\mathbf{w} = [w_1, w_2, \ldots, w_d]$ by
$$\mathbf{w}' = [w_1, w_2, \ldots, w_d, \theta]$$
2. Replace every feature vector $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ by
$$\mathbf{x}' = [x_1, x_2, \ldots, x_d, -1]$$

Then, for the new training set and weight vector, we can treat the threshold as 0 and use the algorithm of Section 12.2.1. The justification is that $\mathbf{w}'.\mathbf{x}' > 0$ is equivalent to $\sum_{i=1}^{d} w_i x_i + \theta \times -1 = \mathbf{w}.\mathbf{x} - \theta > 0$, which in turn is equivalent to $\mathbf{w}.\mathbf{x} > \theta$. The latter is the condition for a positive response from a perceptron with threshold θ.

We can also apply the Winnow Algorithm to the modified data. Winnow requires all feature vectors to have 0's and 1's, as components. However, we can allow a -1 in the feature vector component for θ if we treat it in the manner opposite to the way we treat components that are 1. That is, if the training example is positive, and we need to increase the other weights, we instead divide the component for the threshold by 2. And if the training example is negative, and we need to decrease the other weights we multiply the threshold component by 2.

	and	viagra	the	of	nigeria	θ	y
a	1	1	0	1	1	-1	$+1$
b	0	0	1	1	0	-1	-1
c	0	1	1	0	0	-1	$+1$
d	1	0	0	1	0	-1	-1
e	1	0	1	0	1	-1	$+1$
f	1	0	1	1	0	-1	-1

Figure 12.8: Training data for spam emails, with a sixth component representing the negative of the threshold

Example 12.6: Let us modify the training set of Fig. 12.6 to incorporate a sixth "word" that represents the negative $-\theta$ of the threshold. The new data is shown in Fig. 12.8.

x	y	**w.x**	OK?	and	viagra	the	of	nigeria	θ
				1	1	1	1	1	1
a	$+1$	3	yes						
b	-1	1	no	1	1	$\frac{1}{2}$	$\frac{1}{2}$	1	2
c	$+1$	$-\frac{1}{2}$	no	1	2	1	$\frac{1}{2}$	1	1
d	-1	$\frac{1}{2}$	no	$\frac{1}{2}$	2	1	$\frac{1}{4}$	1	2

Figure 12.9: Sequence of updates to **w** performed by the Winnow Algorithm on the training set of Fig. 12.8

We begin with a weight vector **w** with six 1's, as shown in the first line of Fig. 12.9. When we compute $\mathbf{w}.\mathbf{a} = 3$, using the first feature vector **a**, we are happy because the training example is positive, and so is the dot product. However, for the second training example, we compute $\mathbf{w}.\mathbf{b} = 1$. Since the

example is negative and the dot product is positive, we must adjust the weights. Since **b** has 1's in the third and fourth components, the 1's in the corresponding components of **w** are replaced by 1/2. The last component, corresponding to θ, must be doubled. These adjustments give the new weight vector $[1, 1, \frac{1}{2}, \frac{1}{2}, 1, 2]$ shown in the third line of Fig. 12.9.

The feature vector **c** is a positive example, but $\mathbf{w}.\mathbf{c} = -\frac{1}{2}$. Thus, we must double the second and third components of **w**, because **c** has 1 in the corresponding components, and we must halve the last component of **w**, which corresponds to θ. The resulting $\mathbf{w} = [1, 2, 1, \frac{1}{2}, 1, 1]$ is shown in the fourth line of Fig. 12.9. Next, **d** is a negative example. Since $\mathbf{w}.\mathbf{d} = \frac{1}{2}$, we must again adjust weights. We halve the weights in the first and fourth components and double the last component, yielding $\mathbf{w} = [\frac{1}{2}, 2, 1, \frac{1}{4}, 1, 2]$. Now, all positive examples have a positive dot product with the weight vector, and all negative examples have a negative dot product, so there are no further changes to the weights.

The designed perceptron has a threshold of 2. It has weights 2 and 1 for "viagra" and "nigeria" and smaller weights for "and" and "of." It also has weight 1 for "the," which suggests that "the" is as indicative of spam as "nigeria," something we doubt is true. Nevertheless, this perceptron does classify all examples correctly. □

12.2.5 Multiclass Perceptrons

There are several ways in which the basic idea of the perceptron can be extended. We shall discuss transformations that enable hyperplanes to serve as nonlinear boundaries in the next section. Here, we look at how perceptrons can be used to classify data into many classes.

Suppose we are given a training set with labels in k different classes. Start by training a perceptron for each class; these perceptrons should each have the same threshold θ. That is, for class i treat a training example $(\mathbf{x}, i)$ as a positive example, and all examples $(\mathbf{x}, j)$, where $j \neq i$, as a negative example. Suppose that the weight vector of the perceptron for class i is determined to be $\mathbf{w}_i$ after training.

Given a new vector $\mathbf{x}$ to classify, we compute $\mathbf{w}_i.\mathbf{x}$ for all $i = 1, 2, \ldots, k$. We take the class of $\mathbf{x}$ to be the value of i for which $\mathbf{w}_i.\mathbf{x}$ is the maximum, provided that value is at least θ. Otherwise, $\mathbf{x}$ is assumed not to belong to any of the k classes.

For example, suppose we want to classify Web pages into a number of topics, such as sports, politics, medicine, and so on. We can represent Web pages by a vector with 1 for each word present in the page and 0 for words not present (of course we would only visualize the pages that way; we wouldn't construct the vectors in reality). Each topic has certain words that tend to indicate that topic. For instance, sports pages would be full of words like "win," "goal," "played," and so on. The weight vector for that topic would give higher weights to the words that characterize that topic.

A new page could be classified as belonging to the topic that gives the highest score when the dot product of the page's vector and the weight vectors for the topics are computed. An alternative interpretation of the situation is to classify a page as belonging to all those topics for which the dot product is above some threshold.

12.2.6 Transforming the Training Set

While a perceptron must use a linear function to separate two classes, it is possible to transform the vectors of a training set before applying a perceptron-based algorithm to separate the classes. It is, in principle, always possible to find such a transformation, as long as we are willing to transform to a higher-dimensional space. But if we understand enough about our data, we can often find a simple transformation that works. An example should give the basic idea.

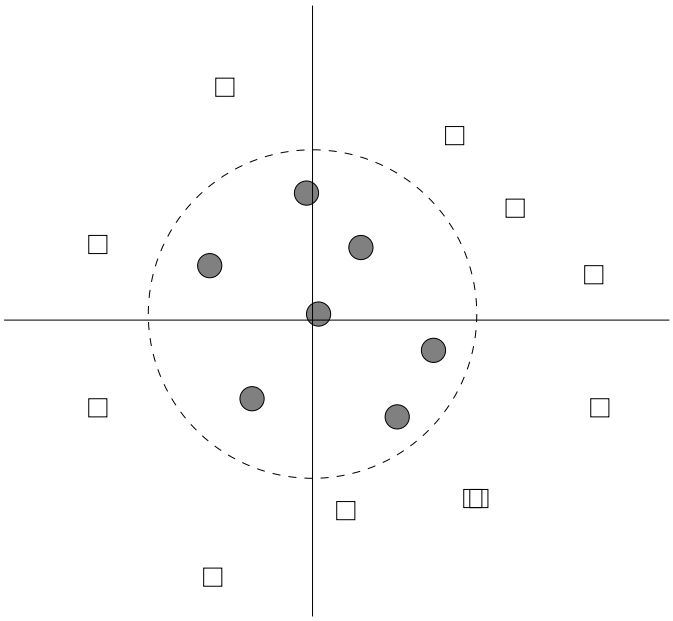


Figure 12.10: Transforming from rectangular to polar coordinates turns this training set into one with a separating hyperplane

Example 12.7: In Fig. 12.10 we see a plot of places to visit from my home. The horizontal and vertical coordinates represent latitude and longitude of places. Some of the places have been classified into "day trips" – places close enough to visit in one day – and "excursions," which require more than a day to visit. These are the circles and squares, respectively. Evidently, there is no

straight line that separates day trips from excursions. However, if we replace the Cartesian coordinates by polar coordinates, then in the transformed space of polar coordinates, the dashed circle shown in Fig. 12.10 becomes a hyperplane. Formally, we transform the vector $\mathbf{x} = [x_1, x_2]$ into $[\sqrt{x_1^2 + x_2^2}, \arctan(x_2/x_1)]$.

In fact, we can also do dimensionality reduction of the data. The angle of the point is irrelevant, and only the radius $\sqrt{x_1^2 + x_2^2}$ matters. Thus, we can turn the point vectors into one-component vectors giving the distance of the point from the origin. Associated with the small distances will be the class label "day trip," while the larger distances will all be associated with the label "excursion." Training the perceptron is extremely easy. □

12.2.7 Problems With Perceptrons

Despite the extensions discussed above, there are some limitations to the ability of perceptrons to classify some data. The biggest problem is that sometimes the data is inherently not separable by a hyperplane. An example is shown in Fig. 12.11. In this example, points of the two classes mix near the boundary so that any line through the points will have points of both classes on at least one of the sides.

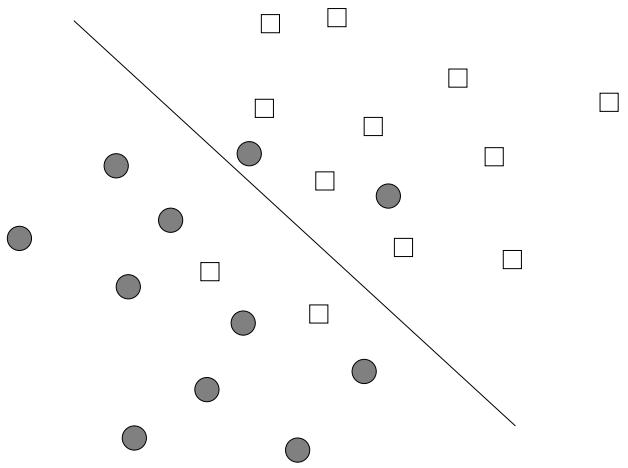


Figure 12.11: A training set may not allow the existence of any separating hyperplane

AS mentioned in Section 12.2.6, it is, in principle, possible to find some function on the points that transforms them to another space where they are linearly separable. However, doing so could well lead to overfitting, the situation where the classifier works very well on the training set, because it has been carefully designed to handle each training example correctly. However, because the classifier is exploiting details of the training set that do not apply to other

examples that must be classified in the future, the classifier will not perform well on new data.

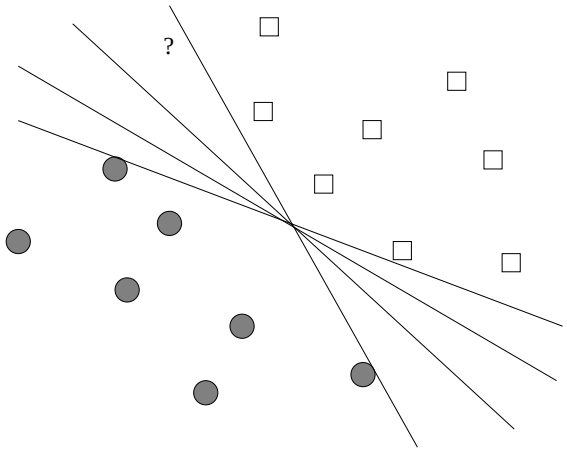


Figure 12.12: Generally, more that one hyperplane can separate the classes if they can be separated at all

Another problem is illustrated in Fig. 12.12. Usually, if classes can be separated by one hyperplane, then there are many different hyperplanes that will separate the points. However, not all hyperplanes are equally good. For instance, if we choose the hyperplane that is furthest clockwise, then the point indicated by "?" will be classified as a circle, even though we intuitively see it as closer to the squares. When we meet support-vector machines in Section 12.3, we shall see that there is a way to insist that the hyperplane chosen be the one that in a sense divides the space most fairly.

Yet another problem is illustrated by Fig. 12.13. Most rules for training a perceptron stop as soon as there are no misclassified points. As a result, the chosen hyperplane will be one that just manages to classify some of the points correctly. For instance, the upper line in Fig. 12.13 has just managed to accommodate two of the squares, and the lower line has just managed to accommodate two of the circles. If either of these lines represent the final weight vector, then the weights are biased toward one of the classes. That is, they correctly classify the points in the training set, but the upper line would classify new squares that are just below it as circles, while the lower line would classify circles just above it as squares. Again, a more equitable choice of separating hyperplane will be shown in Section 12.3.

12.2.8 Parallel Implementation of Perceptrons

The training of a perceptron is an inherently sequential process. If the number of dimensions of the vectors involved is huge, then we might obtain some

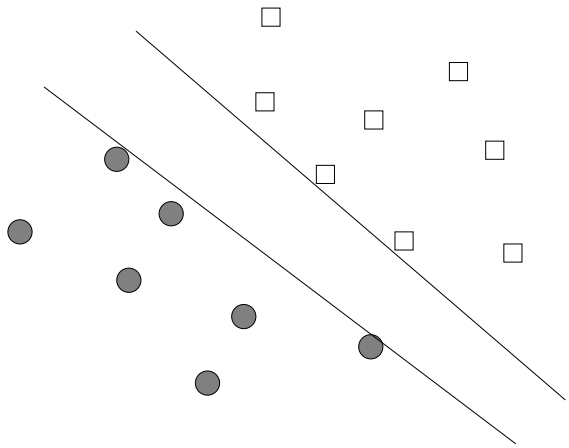


Figure 12.13: Perceptrons converge as soon as the separating hyperplane reaches the region between classes

parallelism by computing dot products in parallel. However, as we discussed in connection with Example 12.4, high-dimensional vectors are likely to be sparse and can be represented more succinctly than would be expected from their length.

In order to get significant parallelism, we have to modify the perceptron algorithm slightly, so that many training examples (a "batch") are used with the same estimated weight vector $\mathbf{w}$. This algorithm modification changes slightly what the algorithm does, when compared with the sequential implementation where we change $\mathbf{w}$ after every misclassified training example. However, if the learning rate is small, as it normally is, then reusing a single $\mathbf{w}$ for many training examples makes little difference in the resulting value of $\mathbf{w}$ after the batch of training examples are considered. As an example, let us formulate the parallel algorithm as a MapReduce job.

The Map Function: Each Map task is given a chunk of training examples, and each Map task knows the current weight vector $\mathbf{w}$. The Map task computes $\mathbf{w}.\mathbf{x}$ for each feature vector $\mathbf{x} = [x_1, x_2, \ldots, x_k]$ in its chunk and compares that dot product with the label y, which is $+1$ or -1, associated with $\mathbf{x}$. If the signs agree, no key-value pairs are produced for this training example. However, if the signs disagree, then for each nonzero component x_i of $\mathbf{x}$ the key-value pair $(i, \eta y x_i)$ is produced; here, η is the learning-rate constant used to train this perceptron. Notice that $\eta y x_i$ is the increment we would like to add to the current ith component of $\mathbf{w}$, and if $x_i = 0$, then there is no need to produce a key-value pair. In the interests of parallelism, we defer that change until we can accumulate many changes in the Reduce phase.

The Reduce Function: For each key i, the Reduce task that handles key i adds all the associated increments and then adds that sum to the ith component

Perceptrons on Streaming Data

While we have viewed the training set as stored data, available for repeated use on any number of passes, perceptrons can also be used in a stream setting. That is, we may suppose there is an infinite sequence of training examples, but that each may be used only once. Detecting email spam is a good example of a training stream. Users report spam emails and also report emails that were classified as spam but are not. Each email, as it arrives, is treated as a training example, and modifies the current weight vector, presumably by a very small amount.

If the training set is a stream, we never really converge, and in fact the data points may well not be linearly separable. However, at all times, we have an approximation to the best possible separator. Moreover, if the examples in the stream evolve over time, as would be the case for email spam, then we have an approximation that values recent examples more than examples from the distant past, much like the exponentially decaying windows technique from Section 4.7.

of $\mathbf{w}$.

Probably, these changes will not be enough to train the perceptron. If any changes to $\mathbf{w}$ occur, then we need to start a new MapReduce job that does the same thing, perhaps with different chunks from the training set. However, even if the entire training set was used on the first round, it can be used again, since its effect on $\mathbf{w}$ will be different if $\mathbf{w}$ has changed.

12.2.9 Exercises for Section 12.2

Exercise 12.2.1: Modify the training set of Fig. 12.6 so that example **b** also includes the word "nigeria" (yet remains a negative example – perhaps someone telling about their trip to Nigeria). Find a weight vector that separates the positive and negative examples, using:

(a) The basic training method of Section 12.2.1.

(b) The Winnow method of Section 12.2.3.

(c) The basic method with a variable threshold, as suggested in Section 12.2.4.

(d) The Winnow method with a variable threshold, as suggested in Section 12.2.4.

! Exercise 12.2.2: For the following training set:

$$([1,2],+1) \quad ([2,1],+1)$$
$$([2,3],-1) \quad ([3,2],-1)$$

A Key Trick to Obtain Parallelism

The method we used in Section 12.2.8 to turn an inherently serial process into a parallel one appears frequently when dealing with large datasets. In the serial version of the algorithm, there is a state that changes at every step. In this case, the state is the weight vector $\mathbf{w}$. As long as changes to the state tend to be small at every step, we can fix the state and calculate, in parallel, the changes to this exact state that would be caused by each of the serial steps. After the parallel step, combine the changes to make a new state, and repeat the parallel step until convergence.

describe all the vectors $\mathbf{w}$ and thresholds θ such that the hyperplane (really a line) defined by $\mathbf{w}.\mathbf{x} - \theta = 0$ separates the points correctly.

! **Exercise 12.2.3:** Suppose the following four examples constitute a training set:

$$([1,2],-1) \quad ([2,3],+1)$$
$$([2,1],+1) \quad ([3,2],-1)$$

(a) What happens when you attempt to train a perceptron to classify these points using 0 as the threshold?

!! (b) Is it possible to change the threshold and obtain a perceptron that correctly classifies these points?

(c) Suggest a transformation using quadratic polynomials that will transform these points so they become linearly separable.

12.3 Support-Vector Machines

We can view a *support-vector machine*, or SVM, as an improvement on the perceptron that is designed to address the problems mentioned in Section 12.2.7. An SVM selects one particular hyperplane that not only separates the points in the two classes, but does so in a way that maximizes the *margin* – the distance between the hyperplane and the closest points of the training set.

In this section, we begin with a discussion of SVM's for training points that are separable, and we show how to maximize the margin in such a case. We then consider a more complex problem, where the points are not linearly separable. In that case, our goal is different. We need to find a hyperplane that does the best it can in separating the two classes. But "best" is a tricky notion. We shall develop a loss function that penalizes misclassified points, but also one that penalizes to a lesser extent points that are correctly classified but are too close to the separating hyperplane. This matter will be taken up in Section 12.3.3.

12.3.1 The Mechanics of an SVM

The goal of an SVM is to select a hyperplane $\mathbf{w}.\mathbf{x} + b = 0$ that maximizes the distance γ between the hyperplane and any point of the training set.[1] The idea is suggested by Fig. 12.14. There, we see the points of two classes and a hyperplane dividing them.

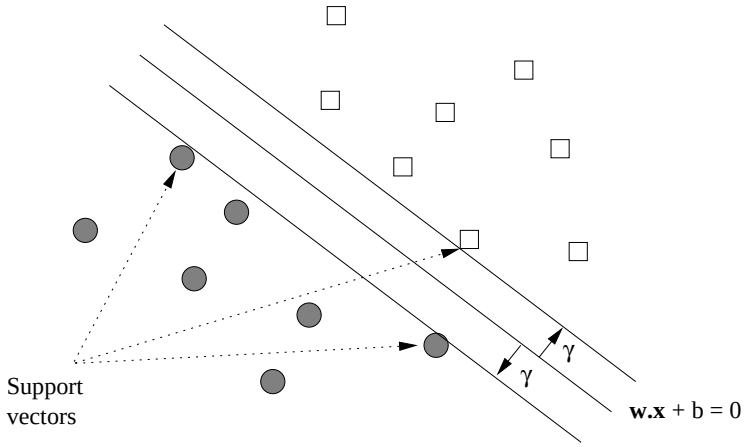


Figure 12.14: An SVM selects the hyperplane with the greatest possible margin γ between the hyperplane and the training points

Intuitively, we are more certain of the class of points that are far from the separating hyperplane than we are of points near to that hyperplane. Thus, it is desirable that all the training points be as far from the hyperplane as possible (but on the correct side of that hyperplane, of course). An added advantage of choosing the separating hyperplane to have as large a margin as possible is that there may be points closer to the hyperplane in the full data set but not in the training set. If so, we have a better chance that these points will be classified properly than if we chose a hyperplane that separated the training points but allowed some points to be very close to the hyperplane itself. In that case, there is a fair chance that a new point that was near a training point that was also near the hyperplane would be misclassified. This issue was discussed in Section 12.2.7 in connection with Fig. 12.13.

We also see in Fig. 12.14 two parallel hyperplanes at distance γ from the central hyperplane $\mathbf{w}.\mathbf{x} + b = 0$, and these each touch one or more of the *support vectors*. The latter are the points that actually constrain the dividing hyperplane, in the sense that they are all at distance γ from the hyperplane. In most cases, a d-dimensional set of points has $d + 1$ support vectors, as is the case in Fig. 12.14. However, there can be more support vectors if too many points happen to lie on the parallel hyperplanes. We shall see an example based

[1] Constant b in this formulation of a hyperplane is the same as the negative of the threshold θ in our treatment of perceptrons in Section 12.2.

on the points of Fig. 11.1, where it turns out that all four points are support vectors, even though two-dimensional data normally has three.

A tentative statement of our goal is:

- Given a training set $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$, maximize γ (by varying $\mathbf{w}$ and b) subject to the constraint that for all $i = 1, 2, \ldots, n$,

$$y_i(\mathbf{w}.\mathbf{x}_i + b) \geq \gamma$$

Notice that y_i, which must be $+1$ or -1, determines which side of the hyperplane the point $\mathbf{x}_i$ must be on, so the $\geq$ relationship to γ is always correct. However, it may be easier to express this condition as two cases: if $y = +1$, then $\mathbf{w}.\mathbf{x}+b \geq \gamma$, and if $y = -1$, then $\mathbf{w}.\mathbf{x} + b \leq -\gamma$.

Unfortunately, this formulation doesn't really work properly. The problem is that by increasing $\mathbf{w}$ and b, we can always allow a larger value of γ. For example, suppose that $\mathbf{w}$ and b satisfy the constraint above. If we replace $\mathbf{w}$ by $2\mathbf{w}$ and b by $2b$, we observe that for all i, $y_i\big((2\mathbf{w}).\mathbf{x}_i + 2b\big) \geq 2\gamma$. Thus, $2\mathbf{w}$ and $2b$ is always a better choice that $\mathbf{w}$ and b, so there is no best choice and no maximum γ.

12.3.2 Normalizing the Hyperplane

The solution to the problem that we described intuitively above is to normalize the weight vector $\mathbf{w}$. That is, the unit of measure perpendicular to the separating hyperplane is the unit vector $\mathbf{w}/\|\mathbf{w}\|$. Recall that $\|\mathbf{w}\|$ is the Frobenius norm, or the square root of the sum of the squares of the components of $\mathbf{w}$. We shall require that $\mathbf{w}$ be such that the parallel hyperplanes that just touch the support vectors are described by the equations $\mathbf{w}.\mathbf{x}+b = +1$ and $\mathbf{w}.\mathbf{x}+b = -1$, as suggested by Fig. 12.15. We shall refer to the hpyerplanes defined by these two equations as the *upper* and *lower* hyperplanes, respectively.

It looks like we have set $\gamma = 1$. But since we are using $\mathbf{w}$ as the unit vector, the margin γ is the number of "units," that is, steps in the direction $\mathbf{w}$ needed to go between the separating hyperplane and the parallel hyperplanes. Our goal becomes to maximize γ, which is now the multiple of the unit vector $\mathbf{w}/\|\mathbf{w}\|$ between the separating hyperplane and the upper and lower hyperplanes.

Our first step is to demonstrate that maximizing γ is the same as minimizing $\|\mathbf{w}\|$. consider one of the support vectors, say $\mathbf{x}_2$ shown in Fig. 12.15. Let $\mathbf{x}_1$ be the projection of $\mathbf{x}_2$ onto the upper hyperplane, also as suggested by Fig. 12.15. Note that $\mathbf{x}_1$ need not be a support vector or even a point of the training set. The distance from $\mathbf{x}_2$ to $\mathbf{x}_1$ in units of $\mathbf{w}/\|\mathbf{w}\|$ is 2γ. That is,

$$\mathbf{x}_1 = \mathbf{x}_2 + 2\gamma\frac{\mathbf{w}}{\|\mathbf{w}\|} \tag{12.1}$$

Since $\mathbf{x}_1$ is on the hyperplane defined by $\mathbf{w}.\mathbf{x} + b = +1$, we know that $\mathbf{w}.\mathbf{x}_1 + b = 1$. If we substitute for $\mathbf{x}_1$ using Equation 12.1, we get

$$\mathbf{w}.\Big(\mathbf{x}_2 + 2\gamma\frac{\mathbf{w}}{\|\mathbf{w}\|}\Big) + b = 1$$

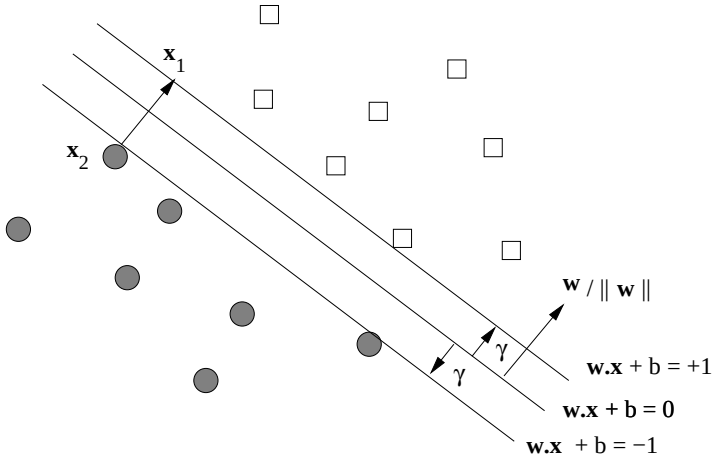


Figure 12.15: Normalizing the weight vector for an SVM

Regrouping terms, we see

$$\mathbf{w}.\mathbf{x}_2 + b + 2\gamma\frac{\mathbf{w}.\mathbf{w}}{\|\mathbf{w}\|} = 1 \tag{12.2}$$

But the first two terms of Equation 12.2, $\mathbf{w}.\mathbf{x}_2 + b$, sum to -1, since we know that $\mathbf{x}_2$ is on the lower hyperplane, $\mathbf{w}.\mathbf{x} + b = -1$. If we move this -1 from left to right in Equation 12.2 and then divide through by 2, we conclude that

$$\gamma\frac{\mathbf{w}.\mathbf{w}}{\|\mathbf{w}\|} = 1 \tag{12.3}$$

Notice also that $\mathbf{w}.\mathbf{w}$ is the sum of the squares of the components of $\mathbf{w}$. That is, $\mathbf{w}.\mathbf{w} = \|\mathbf{w}\|^2$. We conclude from Equation 12.3 that $\gamma = 1/\|\mathbf{w}\|$.

This equivalence gives us a way to reformulate the optimization problem originally stated in Section 12.3.1. Instead of maximizing γ, we want to minimize $\|\mathbf{w}\|$, which is the inverse of γ. That is:

- Given a training set $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$, minimize $\|\mathbf{w}\|$ (by varying $\mathbf{w}$ and b) subject to the constraint that for all $i = 1, 2, \ldots, n$,

$$y_i(\mathbf{w}.\mathbf{x}_i + b) \geq 1$$

Example 12.8: Let us consider the four points of Fig. 11.1, supposing that they alternate as positive and negative examples. That is, the training set consists of

$$\begin{array}{ll} ([1,2],+1) & ([2,1],-1) \\ ([3,4],+1) & ([4,3],-1) \end{array}$$

Let $\mathbf{w} = [u, v]$. Our goal is to minimize $\sqrt{u^2 + v^2}$ subject to the constraints we derive from the four training examples. For the first, where $\mathbf{x}_1 = [1, 2]$ and $y_1 = +1$, the constraint is $(+1)(u + 2v + b) = u + 2v + b \geq 1$. For the second, where $\mathbf{x}_2 = [2, 1]$ and $y_2 = -1$, the constraint is $(-1)(2u + v + b) \geq 1$, or $2u + v + b \leq -1$. The last two points are analogously handled, and the four constraints we derive are:

$$\begin{array}{ll} u + 2v + b \geq 1 & 2u + v + b \leq -1 \\ 3u + 4v + b \geq 1 & 4u + 3v + b \leq -1 \end{array}$$

We shall cover in detail the subject of how one optimizes under constraints; the subject is broad and many packages are available for you to use. Section 12.3.4 discusses one method – gradient descent – in connection with a more general application of SVM, where there is no separating hyperplane. An illustration of how this method works will appear in Example 12.9.

In this simple example, the solution is easy to see: $b = 0$ and $\mathbf{w} = [u, v] = [-1, +1]$. It happens that all four constraints are satisfied exactly; i.e., each of the four points is a support vector. That case is unusual, since when the data is two-dimensional, we expect only three support vectors. However, the fact that the positive and negative examples lie on parallel lines allows all four constraints to be satisfied exactly. □

12.3.3 Finding Optimal Approximate Separators

We shall now consider finding an optimal hyperplane in the more general case, where no matter which hyperplane we choose, there will be some points on the wrong side, and perhaps some points that are on the correct side, but too close to the separating hyperplane itself, so the margin requirement is not met. A typical situation is shown in Fig. 12.16. We see two points that are misclassified; they are on the wrong side of the separating hyperplane $\mathbf{w}.\mathbf{x} + b = 0$. We also see two points that, while they are classified correctly, are too close to the separating hyperplane. We shall call all these points *bad* points.

Each bad point incurs a penalty when we evaluate a possible hyperplane. The amount of the penalty, in units to be determined as part of the optimization process, is shown by the arrow leading to the bad point from the hyperplane on the wrong side of which the bad point lies. That is, the arrows measure the distance from the hyperplane $\mathbf{w}.\mathbf{x} + b = 1$ or $\mathbf{w}.\mathbf{x} + b = -1$. The former is the baseline for training examples that are supposed to be above the separating hyperplane (because the label y is $+1$), and the latter is the baseline for points that are supposed to be below (because $y = -1$).

We have many options regarding the exact formula that we wish to minimize. Intuitively, we want $\|\mathbf{w}\|$ to be as small as possible, as we discussed in Section 12.3.2. But we also want the penalties associated with the bad points to be as small as possible. The most common form of a tradeoff is expressed by a formula that involves the term $\|\mathbf{w}\|^2/2$ and another term that involves a constant times the sum of the penalties.

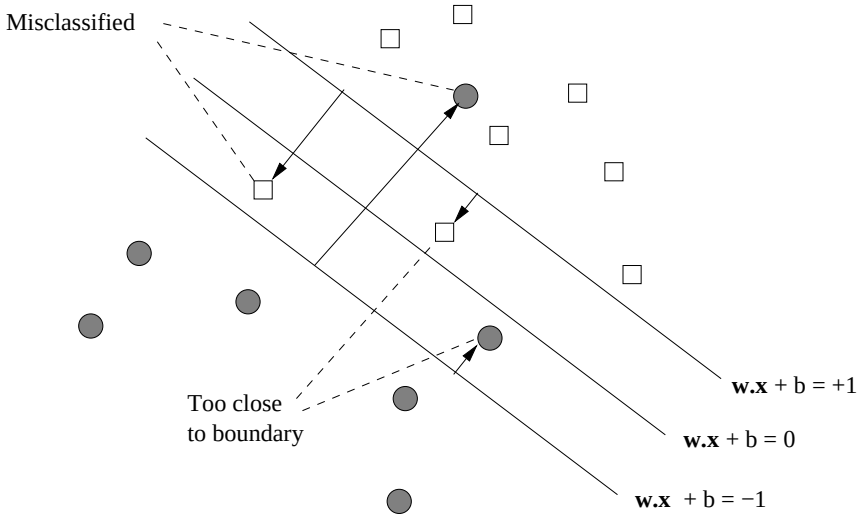


Figure 12.16: Points that are misclassified or are too close to the separating hyperplane incur a penalty; the amount of the penalty is proportional to the length of the arrow leading to that point

To see why minimizing the term $\|\mathbf{w}\|^2/2$ makes sense, note that minimizing $\|\mathbf{w}\|$ is the same as minimizing any monotone function of $\|\mathbf{w}\|$, so it is at least an option to choose a formula in which we try to minimize $\|\mathbf{w}\|^2/2$. This expression is desirable because its derivative with respect to any component of $\mathbf{w}$ is that component. That is, if $\mathbf{w} = [w_1, w_2, \ldots, w_d]$, then $\|\mathbf{w}\|^2/2$ is $\frac{1}{2}\sum_{i=1}^{n} w_i^2$, so its partial derivative $\partial/\partial w_i$ is w_i. This situation makes sense because, as we shall see, the derivative of the penalty term with respect to w_i is a constant times each x_i, the corresponding component of each feature vector whose training example incurs a penalty. That in turn means that the vector $\mathbf{w}$ and the vectors of the training set are commensurate in the units of their components.

Thus, we shall consider how to minimize the particular function

$$f(\mathbf{w}, b) = \frac{1}{2}\sum_{j=1}^{d} w_i^2 + C\sum_{i=1}^{n} \max\Big\{0, \quad 1 - y_i\big(\sum_{j=1}^{d} w_j x_{ij} + b\big)\Big\} \tag{12.4}$$

The first term encourages small $\|\mathbf{w}\|$, while the second term, involving the constant C that must be chosen properly, represents the penalty for bad points in a manner to be explained below. We assume there are n training examples $(\mathbf{x}_i, y_i)$ for $i = 1, 2, \ldots, n$, and $\mathbf{x}_i = [x_{i1}, x_{i2}, \ldots, x_{id}]$. Also, as before, $\mathbf{w} = [w_1, w_2, \ldots, w_d]$. Note that the two summations $\sum_{j=1}^{d}$ express the dot product of vectors.

The constant C, called the *regularization parameter*, reflects how important misclassification is. Pick a large C if you really do not want to misclassify

points, but you would accept a narrow margin. Pick a small C if you are OK with some misclassified points, but want most of the points to be far away from the boundary (i.e., the margin is large).

We must explain the penalty function (second term) in Equation 12.4. The summation over i has one term

$$L(\mathbf{x}_i, y_i) = \max\Big\{0, \quad 1 - y_i\big(\sum_{j=1}^{d} w_j x_{ij} + b\big)\Big\}$$

for each training example $\mathbf{x}_i$. L is the *hinge function*, suggested in Fig. 12.17, and we call its value the *hinge loss*. Let $z_i = y_i(\sum_{j=1}^{d} w_j x_{ij} + b)$. When z_i is 1 or more, the value of L is 0. But for smaller values of z_i, L rises linearly as z_i decreases.

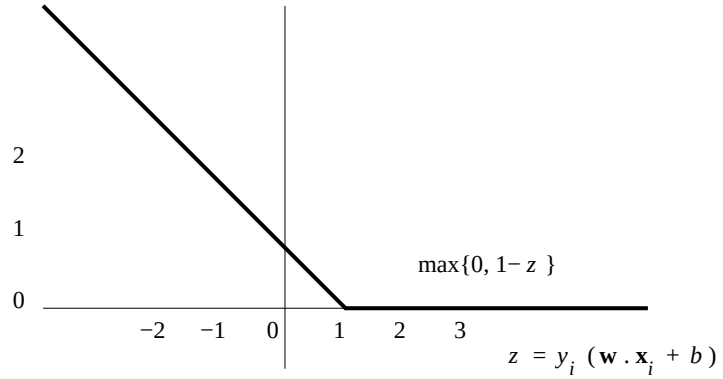


Figure 12.17: The hinge function decreases linearly for $z \leq 1$ and then remains 0

Since we shall have need to take the derivative with respect to each w_j of $L(\mathbf{x}_i, y_i)$, note that the derivative of the hinge function is discontinuous. It is $-y_i x_{ij}$ for $z_i < 1$ and 0 for $z_i > 1$. That is, if $y_i = +1$ (i.e., the ith training example is positive), then

$$\frac{\partial L}{\partial w_j} = \textbf{if } \sum_{j=1}^{d} w_j x_{ij} + b \geq 1 \textbf{ then } 0 \textbf{ else } -x_{ij}$$

Moreover, if $y_i = -1$ (i.e., the ith training example is negative), then

$$\frac{\partial L}{\partial w_j} = \textbf{if } \sum_{j=1}^{d} w_j x_{ij} + b \leq -1 \textbf{ then } 0 \textbf{ else } x_{ij}$$

The two cases can be summarized as one, if we involve the value of y_i, as:

$$\frac{\partial L}{\partial w_j} = \textbf{if } y_i(\sum_{j=1}^{d} w_j x_{ij} + b) \geq 1 \textbf{ then } 0 \textbf{ else } -y_i x_{ij} \tag{12.5}$$

12.3.4 SVM Solutions by Gradient Descent

A common approach to solving Equation 12.4 is to use quadratic programming. For large-scale data, another approach, *gradient descent* has an advantage. We can allow the data to reside on disk, rather than keeping it all in memory, which is normally required for quadratic solvers. To implement gradient descent, we compute the derivative of the equation with respect to b and each component w_j of the vector $\mathbf{w}$. Since we want to minimize $f(\mathbf{w}, b)$, we move b and the components w_j in the direction opposite to the direction of the gradient. The amount we move each component is proportional to the derivative with respect to that component.

Our first step is to use the trick of Section 12.2.4 to make b part of the weight vector $\mathbf{w}$. Notice that b is really the negative of a threshold on the dot product $\mathbf{w}.\mathbf{x}$, so we can append a $(d+1)$st component b to $\mathbf{w}$ and append an extra component with value $+1$ to every feature vector in the training set (not -1 as we did in Section 12.2.4).

We must choose a constant η to be the fraction of the gradient that we move $\mathbf{w}$ in each round. That is, we assign

$$w_j := w_j - \eta \frac{\partial f}{\partial w_j}$$

for all $j = 1, 2, \ldots, d+1$.

The derivative $\frac{\partial f}{\partial w_j}$ of the first term in Equation 12.4, $\frac{1}{2}\sum_{j=1}^{d} w_i^2$, is easy; it is w_j. However, the second term involves the hinge function, so it is harder to express. We shall use an if-then expression to describe these derivatives, as in Equation 12.5. That is:

$$\frac{\partial f}{\partial w_j} = w_j + C \sum_{i=1}^{n} \Big(\textbf{if } y_i(\sum_{j=1}^{d} w_j x_{ij} + b) \geq 1 \textbf{ then } 0 \textbf{ else } -y_i x_{ij} \Big) \qquad (12.6)$$

Note that this formula gives us a partial derivative with respect to each component of $\mathbf{w}$, including w_{d+1}, which is b, as well as to the weights $w_1, w_2, \ldots, w_d$. We continue to use b instead of the equivalent w_{d+1} in the if-then condition to remind us of the form in which the desired hyperplane is described.

To execute the gradient-descent algorithm on a training set, we pick:

1. Values for the parameters C and η.
2. Initial values for $\mathbf{w}$, including the $(d+1)$st component b.

Then, we repeatedly:

(a) Compute the partial derivatives of $f(\mathbf{w}, b)$ with respect to the w_j's.

(b) Adjust the values of $\mathbf{w}$ by subtracting $\eta \frac{\partial f}{\partial w_j}$ from each w_j.

Example 12.9 : Figure 12.18 shows six points, three positive and three negative. We expect that the best separating line will be horizontal, and the only question is whether or not the separating hyperplane and the scale of $\mathbf{w}$ will cause the point $(2, 2)$ to be misclassified or to lie too close to the boundary. Initially, we shall choose $\mathbf{w} = [0, 1]$, a vertical vector with a scale of 1, and we shall choose $b = -2$. As a result, we see in Fig. 12.18 that the point $(2, 2)$ lies on the initial hyperplane and the three negative points are right at the margin. The parameter values we shall choose for gradient descent are $C = 0.1$, and $\eta = 0.2$.

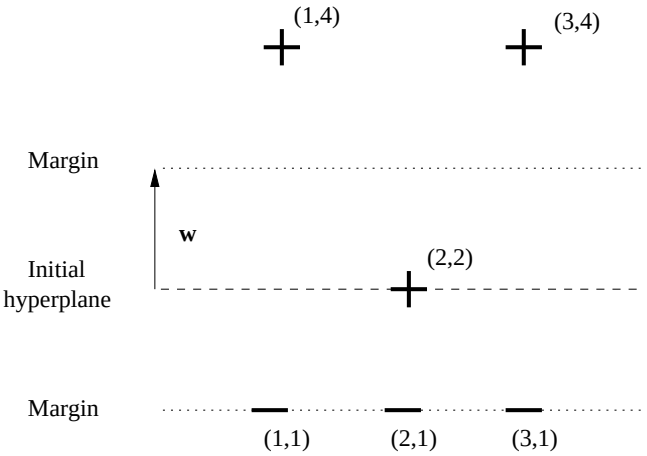


Figure 12.18: Six points for a gradient-descent example

We begin by incorporating b as the third component of $\mathbf{w}$, and for notational convenience, we shall use u and v as the first two components, rather than the customary w_1 and w_2. That is, we take $\mathbf{w} = [u, v, b]$. We also expand the two-dimensional points of the training set with a third component that is always 1. That is, the training set becomes

$$\begin{array}{lll} ([1, 4, 1], +1) & ([2, 2, 1], +1) & ([3, 4, 1], +1) \\ ([1, 1, 1], -1) & ([2, 1, 1], -1) & ([3, 1, 1], -1) \end{array}$$

In Fig. 12.19 we tabulate the if-then conditions and the resulting contributions to the summations over i in Equation 12.6. The summation must be multiplied by C and added to u, v, or b, as appropriate, to implement Equation 12.6.

The truth or falsehood of each of the six conditions in Fig. 12.19 determines the contribution of the terms in the summations over i in Equation 12.6. We shall represent the status of each condition by a sequence of x's and o's, with x representing a condition that does not hold and o representing one that does. The first few iterations of gradient descent are shown in Fig. 12.20.

					for u	for v	for b
if	$u + 4v + b \geq +1$	**then**	0	**else**	-1	-4	-1
if	$2u + 2v + b \geq +1$	**then**	0	**else**	-2	-2	-1
if	$3u + 4v + b \geq +1$	**then**	0	**else**	-3	-4	-1
if	$u + v + b \leq -1$	**then**	0	**else**	$+1$	$+1$	$+1$
if	$2u + v + b \leq -1$	**then**	0	**else**	$+2$	$+1$	$+1$
if	$3u + v + b \leq -1$	**then**	0	**else**	$+3$	$+1$	$+1$

Figure 12.19: Sum each of these terms and multiply by C to get the contribution of bad points to the derivatives of f with respect to u, v, and b

	$\mathbf{w} = [u, v]$	b	Bad	$\partial/\partial u$	$\partial/\partial v$	$\partial/\partial b$
(1)	$[0.000, 1.000]$	-2.000	oxoooo	-0.200	0.800	-2.100
(2)	$[0.040, 0.840]$	-1.580	oxoxxx	0.440	0.940	-1.380
(3)	$[-0.048, 0.652]$	-1.304	oxoxxx	0.352	0.752	-1.104
(4)	$[-0.118, 0.502]$	-1.083	xxxxxx	-0.118	-0.198	-1.083
(5)	$[-0.094, 0.542]$	-0.866	oxoxxx	0.306	0.642	-0.666
(6)	$[-0.155, 0.414]$	-0.733	xxxxxx			

Figure 12.20: Beginning of the process of gradient descent

Consider line (1). It shows the initial value of $\mathbf{w} = [0, 1]$, as we suggested in Fig. 12.18 was the initial value of $\mathbf{w}$. Recall that we use u and v for the components of $\mathbf{w}$, so $u = 0$ and $v = 1$. We also see the initial value of $b = -2$, which is the appropriate value for the initial hyperplane shown in Fig. 12.18. We must use these values of u, v, and b to evaluate the conditions in Fig. 12.19. The first of the conditions in Fig. 12.19 is $u + 4v + b \geq +1$. The left side is $0 + 4 + (-2) = 2$, so the condition is satisfied. However, the second condition, $2u+2v+b \geq +1$ fails. The left side is $0+2+(-2) = 0$. The fact that the sum is 0 means the second point $(2, 2)$ is exactly on the separating hyperplane, and not outside the margin. The third condition is satisfied, since $0+4+(-2) = 2 \geq +1$. The last three conditions are also satisfied, and in fact are satisfied exactly. For instance, the fourth condition is $u+v+b \leq -1$. The left side is $0+1+(-2) = -1$. Thus, the pattern oxoooo represents the outcome of these six conditions, as we see in the first line of Fig. 12.20.

We use these conditions to compute the partial derivatives. For $\partial f/\partial u$, we use u in place of w_j in Equation 12.6. This expression thus becomes

$$u + C\big(0 + (-2) + 0 + 0 + 0 + 0\big) = 0 + \frac{1}{10}(-2) = -0.2$$

The sum multiplying C can be explained this way. For each of the six conditions of Fig. 12.19, take 0 if the condition is satisfied, and take the value in the column labeled "for u" if it is not satisfied. Similarly, for v in place of w_j we

get $\partial f/\partial v = 1 + \frac{1}{10}\big(0 + (-2) + 0 + 0 + 0 + 0\big) = 0.8$. Finally, for b we get $\partial f/\partial b = -2 + \frac{1}{10}\big(0 + (-1) + 0 + 0 + 0 + 0\big) = -2.1$.

We can now compute the new $\mathbf{w}$ and b that appear on line (2) of Fig. 12.20. Since we chose $\eta = 0.2$, the new value of u is $0 - \frac{1}{5}(-0.2) = -0.04$, the new value of v is $1 - \frac{1}{5}(0.8) = 0.84$, and the new value of b is $-2 - \frac{1}{5}(-2.1) = -1.58$.

To compute the derivatives shown in line (2) of Fig. 12.20 we must first check the conditions of Fig. 12.19. While the outcomes of the first three conditions have not changed, the last three are no longer satisfied. For example, the fourth condition is $u + v + b \leq -1$, but $0.04 + 0.84 + (-1.58) = -0.7$, which is not less than -1. Thus, the pattern of bad points becomes oxoxxx. We now have more nonzero terms in the expressions for the derivatives. For example $\partial f/\partial u = 0.04 + \frac{1}{10}\big(0 + (-2) + 0 + 1 + 2 + 3\big) = 0.44$.

The values of $\mathbf{w}$ and b in line (3) are computed from the derivatives of line (2) in the same way as they were computed in line (2). The new values do not change the pattern of bad points; it is still oxoxxx. However, when we repeat the process for line (4), we find that all six conditions are unsatisfied. For instance, the first condition, $u + 4v + b \geq +1$ is not satisfied, because $(-0.118 + 4 \times 0.502 + (-1.083) = 0.807$, which is less than 1. In effect, the first point has become too close to the separating hyperplane, even though it is properly classified.

We can see that in line (5) of Fig. 12.20, the problems with the first and third points are corrected, and we go back to pattern oxoxxx of bad points. However, at line (6), the points have again become too close to the separating hyperplane, so we revert to the xxxxxx pattern of bad points. You are invited to continue the sequence of updates to $\mathbf{w}$ and b for several more iterations.

One might wonder why the gradient-descent process seems to be converging on a solution where at least some of the points are inside the margin, when there is an obvious hyperplane (horizontal, at height 1.5) with a margin of 1/2, that separates the positive and negative points. The reason is that when we picked $C = 0.1$ we were saying that we really don't care too much whether there are points inside the margins, or even if points are misclassified. We were saying also that what was important was a large margin (which corresponds to a small $\|\mathbf{w}\|$), even if some points violated that same margin. □

12.3.5 Stochastic Gradient Descent

The gradient-descent algorithm described in Section 12.3.4 is often called *batch* gradient descent, because at each round, all the training examples are considered as a "batch." While it is effective on small datasets, it can be too time-consuming to execute on a large dataset, where we must visit every training example, often many times before convergence.

An alternative, called *stochastic* gradient descent, considers one training example, or a few training examples at a time and adjusts the current estimate of the error function ($\mathbf{w}$ in the SVM example) in the direction indicated by only the small set of training examples considered. Additional rounds are possible,

using other sets of training examples; these can be selected randomly or according to some fixed strategy. Note that it is normal that some members of the training set are *never* used in a stochastic gradient descent algorithm.

Example 12.10: Recall the UV-decomposition algorithm discussed in Section 9.4.3. This algorithm was described as an example of batch gradient descent. We can regard each of the nonblank entries in the matrix M we are trying to approximate by the product UV as a training example, and the error function is the root-mean-square error between the product of the current matrices U and V and the matrix M, considering only those elements where M is nonblank.

However, if M has a very large number of nonblank entries, as would be the case if M represented, say, purchases of items by Amazon customers or movies that Netflix customers had rated, then it is not practical to make repeated passes over the entire set of nonblank entries of M when adjusting the entries in U and V. A stochastic gradient descent implementation would look at a single nonblank entry of M and compute the change to each element of U and V that would make the product UV agree with that element of M. We would not make that change to the elements of U and V completely, but rather choose some learning rate η less than 1 and change each element of U and V by the fraction η of the amount that would be necessary to make UV equal M in the chosen entry. □

There is a compromise between batch and stochastic gradient descent called *minibatch* gradient descent. In the minibatch version, we partition the entire training set into "minibatches," of some chosen size, e.g., 1000 training examples. We work on one minibatch at a time, computing changes to $\mathbf{w}$ using Equation 12.4, but summing only over the selected training examples.

12.3.6 Parallel Implementation of SVM

The first observation is that stochastic gradient descent is inherently serial, since the state of the system – the vector $\mathbf{w}$ and the constant b – changes with every training example considered. On the other hand, batch gradient descent is most easily parallelized, since it uses each training example starting from the same state, and only combines the effects of these training examples at the end of a round.

Thus, we can parallelize SVM using gradient descent in a manner analogous to what we suggested for perceptrons in Section 12.2.8. You can start with the current $\mathbf{w}$ and b and divide the training examples into minibatches, creating one task for each minibatch. The tasks each apply Equation 12.4 to their minibatch, and the changes to the state, $\mathbf{w}$ and b, are summed after one parallel round. The new state is computed by summing all the changes, and the process can repeat with the new state distributed to all the tasks.

12.3.7 Exercises for Section 12.3

Exercise 12.3.1: Continue the iterations of Fig. 12.20 for three more iterations.

Exercise 12.3.2: The following training set obeys the rule that the positive examples all have vectors whose components sum to 10 or more, while the sum is less than 10 for the negative examples.

$$\begin{array}{lll} ([3,4,5],+1) & ([2,7,2],+1) & ([5,5,5],+1) \\ ([1,2,3],-1) & ([3,3,2],-1) & ([2,4,1],-1) \end{array}$$

(a) Which of these six vectors are the support vectors?

! (b) Suggest a vector $\mathbf{w}$ and constant b such that the hyperplane defined by $\mathbf{w}.\mathbf{x} + b = 0$ is a good separator for the positive and negative examples. Make sure that the scale of $\mathbf{w}$ is such that all points are outside the margin; that is, for each training example $(\mathbf{x}, y)$, you have $y(\mathbf{w}.\mathbf{x} + b) \geq +1$.

! (c) Starting with your answer to part (b), use gradient descent to find the optimum $\mathbf{w}$ and b. Note that if you start with a separating hyperplane, and you scale $\mathbf{w}$ properly, then the second term of Equation 12.4 will always be 0, which simplifies your work considerably.

! **Exercise 12.3.3:** The following training set obeys the rule that the positive examples all have vectors whose components have an odd sum, while the sum is even for the negative examples.

$$\begin{array}{lll} ([1,2],+1) & ([3,4],+1) & ([5,2],+1) \\ ([2,4],-1) & ([3,1],-1) & ([7,3],-1) \end{array}$$

(a) Suggest a starting vector $\mathbf{w}$ and constant b that classifies at least three of the points correctly.

!! (b) Starting with your answer to (a), use gradient descent to find the optimum $\mathbf{w}$ and b.

12.4 Learning from Nearest Neighbors

In this section we consider several examples of "learning" where the entire training set is stored, perhaps preprocessed in some useful way, and then used to classify future examples or to compute the value of the label that is most likely associated with the example. The feature vector of each training example is treated as a data point in some space. When a new point arrives and must be classified, we find the training example or examples that are closest to the new point, according to the distance measure for that space. The estimated label is then computed by combining the closest examples in some way.

12.4.1 The Framework for Nearest-Neighbor Calculations

The training set is first preprocessed and stored. The decisions take place when a new example, called the *query example* arrives and must be classified.

There are several decisions we must make in order to design a nearest-neighbor-based algorithm that will classify query examples. We enumerate them here.

1. What distance measure do we use?
2. How many of the nearest neighbors do we look at?
3. How do we weight the nearest neighbors? Normally, we provide a function (the *kernel function*) of the distance between the query example and its nearest neighbors in the training set, and use this function to weight the neighbors. If there is no weighting, then the kernel function need not be specified.
4. How do we define the label to associate with the query? This label is some function of the labels of the nearest neighbors, perhaps weighted by the kernel function, or perhaps not.

12.4.2 Learning with One Nearest Neighbor

The simplest cases of nearest-neighbor learning are when we choose only the one neighbor that is nearest the query example. In that case, there is no use for weighting the neighbors, so the kernel function is omitted. There is also typically only one possible choice for the labeling function: take the label of the query to be the same as the label of the nearest neighbor.

Example 12.11 : Figure 12.21 shows some of the examples of dogs that last appeared in Fig. 12.1. We have dropped most of the examples for simplicity, leaving only three Chihuahuas, two Dachshunds, and two Beagles. Since the height-weight vectors describing the dogs are two-dimensional, there is a simple and efficient way to construct a *Voronoi diagram* for the points, in which the perpendicular bisectors of the lines between each pair of points is constructed. Each point gets a region around it, containing all the points to which it is the nearest. These regions are always convex, although they may be open to infinity in one direction.[2] It is also a surprising fact that, even though there are $O(n^2)$ perpendicular bisectors for n points, the Voronoi diagram can be found in $O(n \log n)$ time.

In Fig. 12.21 we see the Voronoi diagram for the seven points. The boundaries that separate dogs of different breeds are shown solid, while the boundaries

[2]While the region belonging to any one point is convex, the union of the regions for two or more points might not be convex. Thus, in Fig. 12.21 we see that the region for all Dachshunds and the region for all Beagles are not convex. That is, there are points p_1 and p_2 that are both classified Dachshunds, but the midpoint of the line between p_1 and p_2 is classified as a Beagle, and vice versa.

Figure 12.21: Voronoi diagram for the three breeds of dogs

between dogs of the same breed are shown dashed. Suppose a query example q is provided. Note that q is a point in the space of Fig. 12.21. We find the region into which q falls, and give q the label of the training example to which that region belongs. Note that it is not too hard to find the region of q. We have to determine to which side of certain lines q falls. This process is the same as we used in Sections 12.2 and 12.3 to compare a vector $\mathbf{x}$ with a hyperplane perpendicular to a vector $\mathbf{w}$. In fact, if the lines that actually form parts of the Voronoi diagram are preprocessed properly, we can make the determination in $O(\log n)$ comparisons; it is not necessary to compare q with all of the $O(n \log n)$ lines that form part of the diagram. □

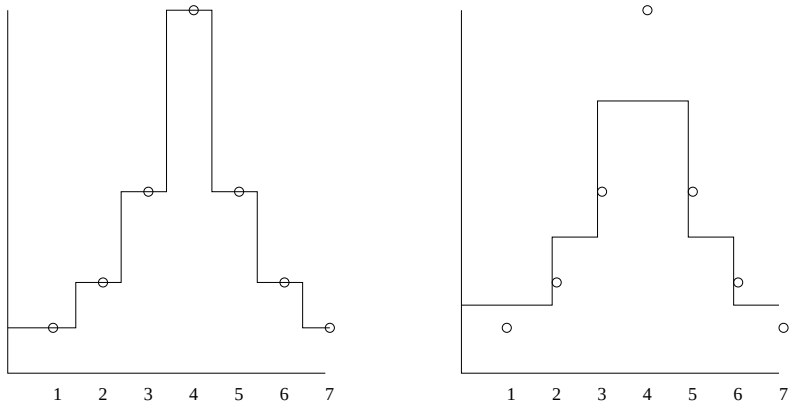
12.4.3 Learning One-Dimensional Functions

Another simple and useful case of nearest-neighbor learning has one-dimensional data. In this situation, the training examples are of the form $([x], y)$, and we shall write them as (x, y), identifying a one-dimensional vector with its lone component. In effect, the training set is a collection of samples of the value of a function $y = f(x)$ for certain values of x, and we must interpolate the function f at all points. There are many rules that could be used, and we shall only outline some of the popular approaches. As discussed in Section 12.4.1, the approaches vary in the number of neighbors they use, whether or not the

neighbors are weighted, and if so, how the weight varies with distance.

Suppose we use a method with k nearest neighbors, and x is the query point. Let $x_1, x_2, \ldots, x_k$ be the k nearest neighbors of x, and let the weight associated with training point (x_i, y_i) be w_i. Then the estimate of the label y for x is $\sum_{i=1}^{k} w_i y_i / \sum_{i=1}^{k} w_i$. Note that this expression gives the weighted average of the labels of the k nearest neighbors.

Example 12.12: We shall illustrate four simple rules, using the training set $(1,1)$, $(2,2)$, $(3,4)$, $(4,8)$, $(5,4)$, $(6,2)$, and $(7,1)$. These points represent a function that has a peak at $x = 4$ and decays exponentially on both sides. Note that this training set has values of x that are evenly spaced. There is no requirement that the points be evenly spaced or have any other regular pattern. Some possible ways to interpolate values are:



(a) One nearest neighbor

(b) Average of two nearest neighbors

Figure 12.22: Results of applying the first two rules in Example 12.12

1. *Nearest Neighbor.* Use only the one nearest neighbor. There is no need for a weighting. Just take the value of any $f(x)$ to be the label y associated with the training-set point nearest to query point x. The result of using this rule on the example training set described above is shown in Fig. 12.22(a).

2. *Average of the Two Nearest Neighbors.* Choose 2 as the number of nearest neighbors to use. The weights of these two are each $1/2$, regardless of how far they are from the query point x. The result of this rule on the example training set is in Fig. 12.22(b).

3. *Weighted Average of the Two Nearest Neighbors.* We again choose two nearest neighbors, but we weight them in inverse proportion to their distance from the query point. Suppose the two neighbors nearest to query

point x are x_1 and x_2. Suppose first that $x_1 < x < x_2$. Then the weight of x_1, the inverse of its distance from x, is $1/(x - x_1)$, and the weight of x_2 is $1/(x_2 - x)$. The weighted average of the labels is

$$\left(\frac{y_1}{x - x_1} + \frac{y_2}{x_2 - x}\right) / \left(\frac{1}{x - x_1} + \frac{1}{x_2 - x}\right)$$

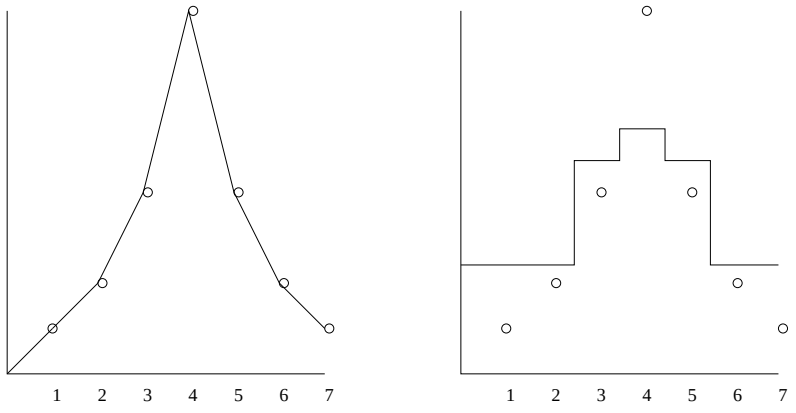
which, when we multiply numerator and denominator by $(x - x_1)(x_2 - x)$, simplifies to

$$\frac{y_1(x_2 - x) + y_2(x - x_1)}{x_2 - x_1}$$

This expression is the linear interpolation of the two nearest neighbors, as shown in Fig. 12.23(a). When both nearest neighbors are on the same side of the query x, the same weights make sense, and the resulting estimate is an *extrapolation*. We see extrapolation in Fig. 12.23(a) in the range $x = 0$ to $x = 1$. In general, when points are unevenly spaced, we can find query points in the interior where both neighbors are on one side.

4. *Average of Three Nearest Neighbors.* We can average any number of the nearest neighbors to estimate the label of a query point. Figure 12.23(b) shows what happens on our example training set when the three nearest neighbors are used, with no weighting. Weighting the three neighbors, such as in inverse proportion to their distances to the point in quesiton, is another option.

□



(a) Weighted average of two neighbors (b) Average of three neighbors

Figure 12.23: Results of applying the last two rules in Example 12.12

12.4.4 Kernel Regression

A way to construct a continuous function that represents the data of a training set well is to consider all points in the training set, but weight the points using a kernel function that decays with distance. A popular choice is to use a normal distribution (or "bell curve"), so the weight of a training point x when the query is q is $e^{-(x-q)^2/\sigma^2}$. Here σ is the standard deviation of the distribution (a parameter you select) and the query q is the mean. Roughly, points within distance σ of q are heavily weighted, and those further away have little weight. There is an advantage to using a kernel function, such as the normal distribution, that is continuous and defined for all points in the training set; doing so assures that the resulting function learned from the data is itself continuous. (See Exercise 12.4.6 for a discussion of the problem when a simpler weighting is used.)

Example 12.13: Let us use the seven training examples of Example 12.12. To make calculation simpler, we shall not use the normal distribution as the kernel function, but rather another continuous function of distance, namely $w = 1/(x-q)^2$. That is, weights decay as the square of the distance. Suppose the query q is 3.5. The weights $w_1, w_2, \ldots w_7$ of the seven training examples $(x_i, y_i) = (i, 8/2^{|i-4|})$ for $i = 1, 2, \ldots, 7$ are shown in Fig. 12.24.

(1)	x_i	1	2	3	4	5	6	7
(2)	y_i	1	2	4	8	4	2	1
(3)	w_i	4/25	4/9	4	4	4/9	4/25	4/49
(4)	$w_i y_i$	4/25	8/9	16	32	16/9	8/25	4/49

Figure 12.24: Weights of points when the query is $q = 3.5$

Lines (1) and (2) of Fig. 12.24 give the seven training points. The weight of each when the query is $q = 3.5$ is given in line (3). For instance, for $x_1 = 1$, the weight $w_1 = 1/(1-3.5)^2 = 1/(-2.5)^2 = 4/25$. Then, line (4) shows each y_i weighted by the weight from line (3). For instance, the column for x_2 has value 8/9 because $w_2 y_2 = 2 \times (4/9)$.

To compute the label for the query $q = 3.5$ we sum the weighted values of the labels in the training set, as given by line (4) of Fig. 12.24; this sum is 51.23. We then divide by the sum of the weights in line (3). This sum is 9.29, so the ratio is $51.23/9.29 = 5.51$. That estimate of the value of the label for $q = 3.5$ seems intuitively reasonable, since q lies midway between two points with labels 4 and 8. □

12.4.5 Dealing with High-Dimensional Euclidean Data

We saw in Section 12.4.2 that the two-dimensional case of Euclidean data is fairly easy. There are several large-scale data structures that have been devel-

Problems in the Limit for Example 12.13

Suppose q is exactly equal to one of the training examples x. If we use the normal distribution as the kernel function, there is no problem with the weight of x; it is 1. However, with the kernel function discussed in Example 12.13, the weight of x is $1/(x-q)^2 = \infty$. Fortunately, this weight appears in both the numerator and denominator of the expression that estimates the label of q. It can be shown that in the limit as q approaches x, the label of x dominates all the other terms in both numerator and denominator, so the estimated label of q is the same as the label of x. That makes excellent sense, since $q = x$ in the limit.

oped for finding near neighbors when the number of dimensions grows, and the training set is large. We shall not cover these structures here, because the subject could fill a book by itself, and there are many places available to learn about these techniques, collectively called *multidimensional index structures*. See the bibliographic notes for this chapter for information about such structures as *kd*-Trees, R-Trees, and Quad Trees.

Unfortunately, for high-dimensional data, there is little that can be done to avoid searching a large portion of the data. This fact is another manifestation of the "curse of dimensionality" from Section 7.1.3. Two ways to deal with the "curse" are the following:

1. *VA Files.* Since we must look at a large fraction of the data anyway in order to find the nearest neighbors of a query point, we could avoid a complex data structure altogether. Accept that we must scan the entire file, but do so in a two-stage manner. First, a summary of the file is created, using only a small number of bits that approximate the values of each component of each training vector. For example, if we use only the high-order (1/4)th of the bits in numerical components, then we can create a file that is (1/4)th the size of the full dataset. However, by scanning this file we can construct a list of candidates that *might* be among the k nearest neighbors of the query q, and this list may be a small fraction of the entire dataset. We then look up only these candidates in the complete file, in order to determine which k are nearest to q.

2. *Dimensionality Reduction.* We may treat the vectors of the training set as a matrix, where the rows are the vectors of the training example, and the columns correspond to the components of these vectors. Apply one of the dimensionality-reduction techniques of Chapter 11, to compress the vectors to a small number of dimensions, small enough that the techniques for multidimensional indexing can be used. Of course, when processing a query vector q, the same transformation must be applied to q before

searching for q's nearest neighbors.

12.4.6 Dealing with Non-Euclidean Distances

To this point, we have assumed that the distance measure is Euclidean. However, most of the techniques can be adapted naturally to an arbitrary distance function d. For instance, in Section 12.4.4 we talked about using a normal distribution as a kernel function. Since we were thinking about a one-dimensional training set in a Euclidean space, we wrote the exponent as $-(x-q)^2$. However, for any distance function d, we can use as the weight of a point x at distance $d(x,q)$ from the query point q the value of

$$e^{-\left(d(x-q)\right)^2/\sigma^2}$$

Note that this expression makes sense if the data is in some high-dimensional Euclidean space and d is the usual Euclidean distance or Manhattan distance or any other distance discussed in Section 3.5.2. It also makes sense if d is Jaccard distance or any other distance measure.

However, for Jaccard distance and the other distance measures we considered in Section 3.5 we also have the option to use locality-sensitive hashing, the subject of Chapter 3. Recall these methods are only approximate, and they could yield false negatives – training examples that were near neighbors to a query but that do not show up in a search.

If we are willing to accept such errors occasionally, we can build the buckets for the training set and keep them as the representation of the training set. These buckets are designed so we can retrieve all (or almost all, since there can be false negatives) training-set points that are have a minimum similarity to a given query q. Equivalently, one of the buckets to which the query hashes will contain all those points within some maximum distance of q. We hope that as many nearest neighbors of q as our method requires will be found among those buckets.

Yet if different queries have radically different distances to their nearest neighbors, all is not lost. We can pick several distances $d_1 < d_2 < d_3 < \cdots$. Build the buckets for locality-sensitive hashing using each of these distances. For a query q, start with the buckets for distance d_1. If we find enough near neighbors, we are done. Otherwise, repeat the search using the buckets for d_2, and so on, until enough nearest neighbors are found.

12.4.7 Exercises for Section 12.4

Exercise 12.4.1: Suppose we modified Example 12.11 to look at the two nearest neighbors of a query point q. Classify q with the common label if those two neighbors have the same label, and leave q unclassified if the labels of the neighbors are different.

(a) Sketch the boundaries of the regions for the three dog breeds on Fig. 12.21.

! (b) Would the boundaries always consist of straight line segments for any training data?

Exercise 12.4.2: Suppose we have the following training set

$$([1,2],+1) \quad ([2,1],-1)$$
$$([3,4],-1) \quad ([4,3],+1)$$

which is the training set used in Example 12.9. If we use nearest-neighbor learning with the single nearest neighbor as the estimate of the label of a query point, which query points are labeled +1?

Exercise 12.4.3: Consider the one-dimensional training set

$$(1,1),\ (2,2),\ (4,3),\ (8,4),\ (16,5),\ (32,6)$$

Describe the function $f(q)$, the label that is returned in response to the query q, when the interpolation used is:

(a) The label of the nearest neighbor.

(b) The average of the labels of the two nearest neighbors.

! (c) The average, weighted by distance, of the two nearest neighbors.

(d) The (unweighted) average of the three nearest neighbors.

! **Exercise 12.4.4:** Apply the kernel function of Example 12.13 to the data of Exercise 12.4.3. For queries q in the range $2 < q < 4$, what is the label of q?

Exercise 12.4.5: What is the function that estimates the label of query points using the data of Example 12.12 and the average of the four nearest neighbors?

!! **Exercise 12.4.6:** Simple weighting functions such as those in Example 12.12 need not define a continuous function. We can see that the constructed functions in Fig. 12.22 and Fig. 12.23(b) are not continuous, but Fig. 12.23(a) is. Does the weighted average of two nearest neighbors always give a continuous function?

12.5 Decision Trees

A decision tree is a branching program that uses properties of a feature vector to produce the class to which that input belongs. We generally show the decisions made in the form of a tree. In this section, we shall discuss how to design trees that correctly classify training data. In a decision tree, each nonleaf node represents a test on the input. Its children are either tests (which we denote by ellipses) or a leaf that is a conclusion about the output (denoted by a rectangle). The children of a test node are labeled by the outcome of the test. Typically,

there are only two outcomes – true or false (yes or no) – but there could be any number of outcomes of a test.

Also in this section, we examine ways to exploit parallelism while looking for the most efficient tree. As overfitting is a common problem, we shall also talk about how to simplify the tree by removing nodes, to reduce overfitting without losing too much accuracy. We also examine ways to exploit parallelism

12.5.1 Using a Decision Tree

Let us begin with an example of some training data and a tree that might be constructed from this data. The table in Fig. 12.25 gives 12 countries, their population (in millions), their continent, and their favorite sport. We shall treat Population and Continent as features of the input vector and the favorite sport as the class, or output. We assume the countries themselves are not known, and we want to predict the favorite sport just from the continent and population. In particular, we want to predict the favorite sport of countries other than these 12, from their continent and population only.

Country	Continent	Population	Sport
Argentina	SA	44	Soccer
Australia	Aus	34	Cricket
Brazil	SA	211	Soccer
Canada	NA	36	Hockey
Cuba	NA	11	Baseball
Germany	Eur	80	Soccer
India	Asia	1342	Cricket
Italy	Eur	59	Soccer
Russia	Asia	143	Hockey
Spain	Eur	46	Soccer
United Kingdom	Eur	65	Cricket
United States	NA	326	Baseball

Figure 12.25: Favorite sports of countries

Example 12.14: In Fig. 12.26 is a tree that correctly classifies the twelve countries of Fig. 12.25. To classify a country, given its population and continent, we start at the root and apply the test at the root. We go to the left child if the outcome of the test is true and to the right child if not. Whenever we are at a nonleaf node, we apply the test at that node and again move to the left or right child, depending on whether or not the test is satisfied. When we reach a leaf, we output the class at that leaf.

You can check that each of the twelve countries is correctly classified by this tree. For example, consider Spain. The continent of Spain is Europe, so the test at the root is satisfied. Thus, we go the the left child of the root, which

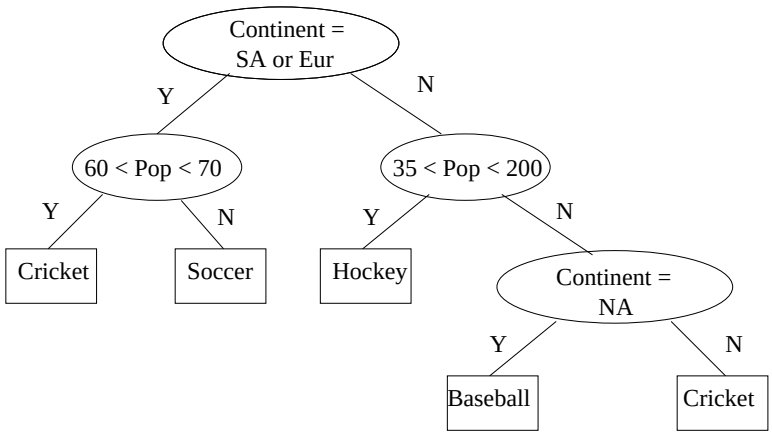


Figure 12.26: A decision tree for the favorite sports of countries

asks us to test whether the population of Spain is between 60 and 70 million. It is not, so we go to the right child, which is a leaf telling us correctly that the favorite sport of Spain is Soccer.

However, there are many countries not in the table of Fig. 12.25, and the decision tree does not do too well on these. For example, consider Pakistan, a country in Asia with a population of 182 million. Starting at the root, we find the condition there not satisfied, since Pakistan is not a European or South-American country. Thus, we go to the right child of the root. Since the population of Pakistan is in the range between 35 and 200 million, we move left, whereupon we are told that the favorite sport in Pakistan is Hockey.

The problem we face is overfitting. That is, the test at the root makes sense; Soccer is most popular in South America and Europe. But tests on population are probably useless, since it is unlikely that the size of a country has anything to do with what sport its people like. We have, in this example, simply used the population to distinguish among the small number of countries in Fig. 12.25 that reach a node of the tree. But unlike the root, the tests at the second level don't really say anything that applies to the larger, unseen set of countries. □

12.5.2 Impurity Measures

To design a decision tree, we need to choose good tests at the various nonleaf nodes of the tree. We would like to use as few levels as possible, so that new data points can be classified quickly, and we have a chance of avoiding the overfitting that we saw in Example 12.14. Ideally, we would like all the inputs that reach a certain node to have the same class, since then we can make that node a leaf and correctly classify all the training examples that reach the node.

We can formalize the property we would like for a node by the notion of *impurity*. There are many impurity measures we could use, but they all have

the property that they are 0 for a node that is reached only by training examples with a single class. Here are three of the most common impurity measures. Each applies to a node reached by training examples with n classes, with p_i being the fraction of those training examples that belong the ith class, for $i = 1, 2, \ldots, n$.

1. *Accuracy*: the fraction of the reaching inputs that are correctly classified, or $1 - \max(p_1, p_2, \ldots, p_n)$.
2. *GINI Impurity*: $1 - \sum_{i=1}^{n} (p_i)^2$.
3. *Entropy*: $\sum_{i=1}^{n} p_i \log_2(1/p_i)$.

Example 12.15: Consider the impurity of the root of Fig. 12.26. There are four classes. Soccer is the favorite sport of 5/12 of the countries in the training data. Baseball and Hockey are each the class of 1/6 of the training examples, while Cricket is the class of 1/4 of the training examples. The impurity of the root according to the accuracy measure is therefore $1 - 5/12 = 7/12 = .583$. The GINI impurity of the root is $1 - (1/6)^2 - (1/6)^2 - (1/4)^2 - (5/12)^2 = 103/144 = 0.715$. The entropy of the root is

$$\frac{1}{6}\log_2(6) + \frac{1}{6}\log_2(6) + \frac{1}{4}\log_2(4) + \frac{5}{12}\log_2(12/5) = 1.875$$

Note the fact that these impurity measures have rather different values is unimportant. These measures have different ranges of possible values; see Exercise 12.5.2.

The left child of the root is much less impure. It is reached by the six countries in South America and Europe, of which five like Soccer and one likes Cricket. The impurity of this node according to the accuracy measure is thus $1 - 5/6 = 1/6 = .167$, while the GINI impurity is $1 - (1/6)^2 - (5/6)^2 = 5/18 = .278$. The entropy of the left child of the root is $\frac{1}{6}\log_2(6) + \frac{5}{6}\log_2(6/5) = .643$. □

12.5.3 Designing a Decision-Tree Node

The goal of node design is to produce children whose weighted average impurity is as small as possible, where the weighting of the children is proportional to the number of training examples that reach the node. In principle, the test at a node could be any function of the input. As this set of possibilities is essentially infinite, we need to restrict ourselves to simple tests at each node. In what follows, we will limit the possible tests to binary decisions based on one of two factors:

1. The comparison of one numerical feature of the input vector with a constant.
2. A test whether one categorical feature of the input vector is in a set of possible values.

Example 12.16: Notice that the tests at the children of the root in Fig. 12.26 do not satisfy condition (1) for numerical features. For example, the right child of the root is the logical AND of two comparisons, population > 55 and population < 200. However, we could use both these conditions if we replaced the single node by two nodes, each testing one of the conditions. The root is a test for membership of the value of feature Continent in the set of two continents SA and Eur, so it does satisfy condition (2). □

Suppose we are given a node of the decision tree that is reached by a subset of the training examples. If the node is pure, i.e., all these training examples have the same output, then we make the node a leaf with that output as value. However, if the impurity is greater than zero, we want to find the test that gives the greatest reduction in impurity. When selecting this test, we are free to choose any feature of the input vector. If we choose a numerical feature, we can choose any constant to divide the training examples into two sets, one going to the left child and the other going to the right child. Alternatively, if we choose a categorical feature, then we may choose any set of values for the membership test. We shall consider each of these cases in turn.

12.5.4 Selecting a Test Using a Numerical Feature

Suppose we want to split a set of training examples based on a numerical feature A. In our running example, A could only be Population, one of the two features – Population and Continent – that are components of feature vectors (recall we assume we do not see the country name; that name is used only to identify each training example). To select the best breakpoint, we

1. Order the training examples according to their value of A. Let the values of A in this order be $a_1, a_2, \ldots, a_n$.

2. For $j = 1, 2, \ldots, n$, compute the number of training examples in each class among $a_1, a_2, \ldots, a_j$. Note that these counts can be done incrementally, since the count for a class after the jth example is either the same as the count for $j - 1$ (if the jth example in not in this class) or one more than the count for $j - 1$ (if the jth example is in this class).

3. From the counts computed in the previous step, compute the weighted-average impurity assuming the test sends the first j training examples to the left child and the remaining $n - j$ examples to the right node. Here, we must assume the impurity can be computed from the counts for each class. That is surely the case for the three measures of impurity – accuracy, GINI, and entropy – that we discussed in Section 12.5.2.

4. Select that value of j that minimizes the weighted-average impurity. However, note that not every possible value of j can be used at this step, since it is possible that $a_j = a_{j+1}$. We need to restrict our choice of j to those

values such that $a_j < a_{j+1}$, so that we can use $A < (a_j + a_{j+1})/2$ as the comparison.

Example 12.17: Suppose we use the root comparison from Fig. 12.26, which sends the six countries from Europe and South America to the left child, and the other six to the right. Now, we need to divide the six countries reaching the left child in such a way that the weighted impurity of the two children of the root's left child is as small as possible. We could use either the Continent or Population feature to do the division, and we must consider both in order to find the split that minimizes the impurity. Here, we shall consider only Population, since that is the only numerical feature. In Fig. 12.27 we see the six countries that reach the left child of the root, ordered by their population.

Ctry.	Pop.	Sp.	n_S	n_C	$p_S\leq$	$p_C\leq$	$p_S>$	$p_C>$	Im$\leq$	Im$>$	Wtd.
Arg.	44	S	1	0	1	0	4/5	1/5	0	8/25	4/15
Spain	46	S	2	0	1	0	3/4	1/4	0	3/8	1/4
Italy	59	S	3	0	1	0	2/3	1/3	0	4/9	2/9
UK	65	C	3	1	3/4	1/4	1	0	3/8	0	1/4
Ger.	80	S	4	1	4/5	1/5	1	0	8/25	0	4/15
Bra.	211	S	5	1	5/6	1/6	–	–	–	–	–

Figure 12.27: Computing the GINI index for each possible separation

The columns explain the calculation as follows. First, the column labeled "Sp" is the favorite sport, which for this set of countries is either Soccer (S) or Cricket (C). The columns n_S and n_C are the cumulative counts of the number of rows with sport Soccer and Cricket, respectively. For instance, in the row for Germany, we see $n_S = 4$, since four of the five rows from Argentina down to Germany have sport Soccer. In that row, we also have $n_C = 1$, since only one of the first five rows has sport Cricket.

The next two columns, labeled $p_S\leq$ and $p_C\leq$ are the fractions of the rows down to and including the row in question, that have sport Soccer and Cricket, respeectively. For example, in the row for Germany, four of the five rows from Argentina to Germany have sport Soccer, so $p_S\leq = 4/5$. The two columns after that, labeled $p_S>$ and $p_C>$, are the same fractions, but for the rows that are below the row in question. For example, in the row for Spain we see $p_S> = 3/4$, since three of the four rows from Italy down to Brazil have sport Soccer, and one of these rows has sport Cricket. Note that we do not have to do a cumulative count working up the rows. We can get the count 3 by subtracting $n_S = 2$ for Spain from the bottom value of n_S (that for Brazil), which is 5. Similarly we know there is one row below Spain with sport Cricket because 1 is the difference between n_C for Brazil and Spain.

Next, we see the columns for impurities, Im$\leq$ and Im$>$. These are the GINI impurities of the left and right children of the node we are designing, assuming that we use a comparison "population $< c$," for some c that lies between the

populations for the row in question and the next row. Thus, in each row, we have Im$\leq$ $= 1 - (p_S\leq)^2 - (p_C\leq)^2$ and Im$>$ $= 1 - (p_S>)^2 - (p_C>)^2$. For instance, in the row for Spain, we have Im$\leq$ $= 1 - 1^2 - 0^2 = 0$ and

$$\text{Im}> \;=\; 1 - (3/4)^2 - (1/4)^2 = 3/8$$

Finally, the last column is the weighted GINI impurity for the two children, weighted by the number of countries that reach each of these children. For example, in the row for Spain, if we divide between the populations of Spain and the next more populous country, Italy, then two countries go to the left child and four to the right child. Thus, the weighted impurity is $(2/6)0+(4/6)(3/8) = 1/4$. Of the five rows after which a split is possible, we find the smallest weighted impurity to be 2/9, which we get by splitting after Italy. That is, we would use a test like "Population $<$ 60" to send Argentina, Spain, and Italy, all of whom like Soccer, to the left. The other three countries, two of which like Soccer and one Cricket, are sent to the right child. Since the left child is pure, it is a leaf declaring Soccer to be the sport, while the right child will need to be split again.

Technically, we also have to consider that the test at the left child of the root does not involve population at all, but rather is a test on the continent. However, we cannot do better by splitting Europe from South America; that gives a weighted GINI impurity of 1/4. □

12.5.5 Selecting a Test Using a Categorical Feature

Now consider how we might split the training examples that reach a node, using the value of a categorical feature A. To avoid having to consider all possible subsets of the values of A, we shall only consider the case where there are two classes. In our running example, we might suppose that the two classes are Soccer (S) and anything but soccer (N). When there are two classes, we can order the values of A by the fraction of training examples with that value belonging to the first class.

Since there are only two classes, the partition of values with the lowest impurity will surely be based on a set of values that is a prefix of this order. That is, this set consists of the values of A whose fractions in the first class are above some threshold. Thus, the examples going to the left child will have many examples in the first class, while the examples going right will have many in the second class.

The process of finding the split in the ordered list of values that minimizes the weighted impurity is then essentially the same as for numerical features, which we explained in Section 12.5.4. The difference is that now we are going down a list of values of A, rather than down a list of training examples. We shall work an example, based on the categorical feture Continent from our running example.

Example 12.18: Let us see how we might design the root of Fig. 12.26, assuming that we recognized only two classes: Soccer (S) and other sports (N).

As before, we shall use GINI as the measure of impurity for nodes. Figure 12.28 summarizes our calculation. In the columns labeled S and N, we see the counts of training examples from Fig. 12.25 for which the favorite sport is Soccer and anything else. The continents are ordered by the fraction of their countries in the Soccer class. Thus, South America is first, because 100% of its training examples are in the class S. Then comes Europe, where 75% of its training examples are in class S. The remaining three continents have 0% in the Soccer class, and could have been ordered in any way.

Cont.	S	N	n_S	n_N	$p_S\leq$	$p_N\leq$	$p_S>$	$p_N>$	Im$\leq$	Im$>$	Wtd.
SA	2	0	2	0	1	0	3/10	7/10	0	21/50	7/20
Eur	3	1	5	1	5/6	1/6	0	1	5/18	0	5/36
NA	0	3	5	4	5/9	4/9	0	1	40/81	0	10/27
Asia	0	2	5	6	5/11	6/11	0	1	60/121	0	5/11
Aus	0	1	5	7	5/12	7/12	–	–	–	–	–

Figure 12.28: Computing the GINI index for sets of continents

The next columns in Fig. 12.28 are n_S and n_N. These are the cumulative sums of the number of examples in class S and not in class S, respectively, counting from the top. For instance, the row for North America has $n_S = 5$ and $n_N = 4$, because among the first three rows there are five examples in the class S and four in the class N. Note that, as for numerical features, we can compute the cumulative sum by a single pass from the top to the bottom. Also, we can get the numbers in each class among all the rows below by subtracting n_S or n_N from its corresponding value at the bottom row.

The next two columns are the fractions in each class for that row and above. That is, $p_S\leq = n_S/(n_S + n_N)$ and $p_N\leq = n_N/(n_S + n_N)$. After those come the two columns $p_S>$ amd $p_N>$ that give the fractions in the classes S and N among all the rows below. For example, in the row for NA, there are zero members of the class S below, which we can see because $n_S = 5$ in both the row for NA and the bottom row. Also, there are 3 members of the class N below, since the value of n_N for row NA is 4, while the value of n_N in the bottom row is 7.

The columns Im$\leq$ and Im$>$ follow. As in Example 12.17, these are the GINI impurities of the left and right children, assuming the row in question and all rows above are sent to the left child and all rows below are sent to the right child. Then, the last column gives the weighted GINI impurity of the children. For instance, consider the entries in the row for NA. If we use the test for whether the Continent is one of SA, Eur, and NA to send examples to the left child, then nine of the training examples will go left, and the remaining three will go right. Thus, the weighted GINI impurity for this split is $(9/12)(40/81) + (3/12)0 = 10/27$.

By far the best split comes after Eur, with a weighted impurity of 5/36. That was the split we used in Fig. 12.26. □

12.5.6 Parallel Design of Decision Trees

The design of decision trees using the methods just described involves a serious amount of computation. The procedure must be applied to every node of the tree. Moreover, while we have described what to do with one feature of the input vector, the same must be done for every feature, after which we pick the best split among all features. If the feature is numerical, we need to sort all the training examples that reach the node. If the feature is categorical, we need to first group the training examples by the value of the feature, and then sort the values according to the fraction of examples that belong to the first class. Moreover, if there are really more than two classes, we need to consider all ways to divide the classes into two groups; these groups then play the role of the two classes in the discussion of Section 12.5.5.

However, to speed the process, there is a lot of easy parallelism available.

- At a node, we can find the best splits for all the features in parallel.
- All the nodes at one level can be designed in parallel. Moreover, each training example reaches at most one node at any level. A training example will reach exactly one node at a given level, unless it reaches a leaf at a higher level. Thus, even without parallelism, we expect the total work at each level to be about the same, rather than growing with the number of nodes.
- Grouping of training examples by value of a feature can be done efficiently in parallel. For example, we discussed how to do this task using MapReduce in Section 2.3.8.
- Parallelism can speed up sorting considerably. While we shall not discuss it here, there are known algorithms that can sort n items in parallel in $O(\log^2 n)$ parallel steps in the worst case, or $O(\log n)$ parallel steps on average.

Suppose now that we are working on the design of one node using one feature A, perhaps in parallel with many other node-feature pairs in parallel. After grouping (if A is nonnumeric) and sorting, we need to compute several accumulated sums. For instance, if A is numeric, then we need to compute for each training example and each class the number of training examples with that class, among this training example and all previous examples on the sorted list. The computation of accumulated sums appears to be inherently serial, but as we shall see, it can be parallelized quite well. Once we have the accumulated sums, we can compute the needed fractions and impurity values associated with each member of the list, in parallel. Notice that the row-by-row calculations suggested in Examples 12.17 and 12.18 are all independent and so open to a parallel implementation.

To complete the story of parallelization for decision trees, let us see how to compute accumulated sums in parallel. Formally, suppose we are given a

list of numbers $a_1, a_2, \ldots, a_n$, and we wish to compute $x_i = \sum_{j=1}^{i} a_j$ for all $i = 1, 2, \ldots, n$. It might appear that we need n steps to compute all the x_i's, and that could be time consuming when n is large; i.e., there is a large training set. However, there is a divide-and-conquer algorithm to calculate all the x's in $O(\log n)$ parallel steps, as follows:

BASIS: If $n = 1$, then $x_1 = a_1$. The basis requires one parallel step.

INDUCTION: If $n > 1$, divide the list of a's into a left half and a right half, as evenly as possible. That is, if n is even, the left half is $a_1, a_2, \ldots, a_{n/2}$, and the right half is $a_{n/2+1}, a_{n/2+2}, \ldots, a_n$. If n is odd, then the left half ends with $a_{\lfloor n/2 \rfloor}$ and the right half begins with $a_{\lceil n/2 \rceil}$.

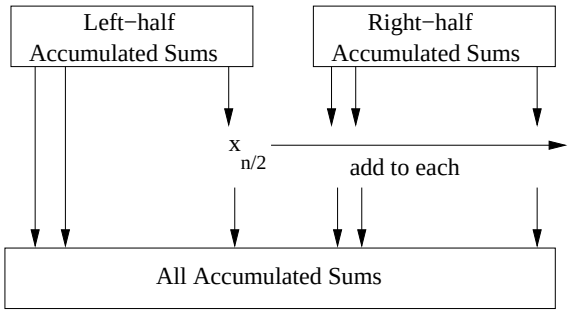


Figure 12.29: Recursive step in parallel computation of accumulated sums

The recursive step is suggested by Fig. 12.29. In parallel, we apply this algorithm to the left and right halves. Once we have done so, the last accumulated sum of the left half, shown as $x_{n/2}$, is then added in parallel to each of the accumulated sums of the right half. As a result, if the ith result in the right half is $\sum_{j=n/2+1}^{i} a_j$, we have, by adding $x_{n/2}$ to it, changed its value to the correct $\sum_{j=1}^{n/2+i} a_j$.

Each time we apply the recursive step, we require only one addition, done in parallel. If n is a power of 2, then each application of the recursive step divides the size of the lists we need to work on by 2, so the total number of parallel steps is $1 + \log_2 n$. If n is not a power of 2, the number of times we perform the recursive step is no greater than what we would need if n were the next higher power of 2. Thus, $1 + \lceil \log_2 n \rceil$ parallel steps suffices for any n.

12.5.7 Node Pruning

If we design a decision tree using as many levels as needed so that each leaf is pure, we are likely to have overfit to the training data. If we can validate our design using other examples – either a test set that we have withheld from the training data or a validation set of new data – we have the opportunity to simplify the tree and at the same time limit the overfitting.

Find a node N that has only leaves as children. Build a new tree by replacing N and its children by a leaf, and give that leaf its majority class as its output. Then, compare the performance of the old and new trees on data that was not used as training examples when we designed the tree. If there is little difference between the error rates of the old and new trees, then the decision made at the node N probably contributed to overfitting and was not addressing a property of the entire set of examples for which the decision tree was intended. We can discard the old tree and replace it by the new, simpler tree. On the other hand, if the new tree has a significantly higher error rate than the old, then the decision made at node N really reflects a property of the data, and we need to retain the old tree and discard the new. In either case, we should continue looking at other nodes whose children are leaves, and see if these can be replaced by leaves without a significant increase in the error rate.

Example 12.19: Let us consider the node in Fig. 12.26 with the test "Continent = NA." It has two leaves as children and so it can play the role of node N above. In the training data of Fig. 12.25 N is reached by three training examples, Cuba, the US, and Australia. As two of these three go to the leaf labeled "Baseball," we shall consider replacing N by a leaf labeled "Baseball."

Consider what happens if the old and new trees are applied to the entire set of countries of the world. First, note that to reach node N, a country has to be in one of the continents North America, Asia, Australia, or Africa. Moreover, it has to be either a small country (less than 35 million population) or a populous country (more than 200 million). There are many such countries, such as small countries in Africa, or large countries like China or Indonesia, none of which have either cricket or baseball as their favorite sport. If we consider the Carribean and Central-American countries as belonging to North America, then N is reached by many more countries, only a few of which have baseball as their favorite sport.

The conclusion is that whether or not N is replaced by a "Baseball" leaf, the tree will have a significant error rate on these countries, and therefore the error rate of the two trees applied to all countries will be about the same. We conclude that this node N reflects only artifacts of the small and atypical set of twelve training examples that we chose. Thus, N can safely be replaced by a leaf without much increase in the error rate on the full set of countries. □

12.5.8 Decision Forests

Because a single decision tree with many levels is likely to have many nodes at the lower levels that represent overfitting, there is another approach to the use of decision trees that has proven quite useful in practice. It is common to use *decision forests* of many trees that vote on the class to which a given data point belongs. Each tree in the forest is designed using randomly or systematically chosen features, and is restricted to only a small number of levels, often one or two. Thus, each tree has a high impurity at each of its leaves, but collectively

Ensemble Methods

Decision forests are but one example of an important strategy for making good decisions. We can use several different algorithms to make the same decision. Then, we combine the opinions of the different algorithms in some way, perhaps by learning the best weights as we suggested in Section 12.5.8. In the case of a decision forest, all the contributing algorithms are of the same type: a decision tree. However, the decision algorithms can also be of different kinds. For example, the winning solution to the Netflix challenge (see Section 9.5) was of this type, where several different machine-learning techniques were each applied, and their movie-rating estimates combined to make a better decision.

they often do much better on test data than any one tree, however many levels it has. Further, we can design each of the trees in the forest in parallel, so it can be even faster to design a large collection of shallow trees than to design one deep tree.

The obvious way to combine the outcomes from all the trees in a decision forest is to take a majority vote. If there are more than two classes, then we only expect a plurality of votes for one of the classes, while no class may be the choice of more than half the trees. However, more complex ways of combining the trees' results may yield a more accurate answer than straightforward voting. We can often "learn" the proper weights to apply to the opinions of each tree.

For example, suppose we have a training set $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)$ and a collection of decision trees $T_1, T_2, \ldots, T_k$ that constitute our decision forest. When we apply the decision forest to one of the training examples $(\mathbf{x}_i, y_i)$, we get a vector of classes $\mathbf{c}_i = [c_{i1}, c_{i2}, \ldots, c_{ik}]$, where c_{ij} is the outcome of tree T_j applied to input $\mathbf{x}_i$. We know the correct class for the input $\mathbf{x}_i$; it is y_i. Thus, we have a new training set $(\mathbf{c}_1, y_1), (\mathbf{c}_2, y_2), \ldots, (\mathbf{c}_n, y_n)$, that we can use to predict the true class from the opinions of all the trees in the forest. For instance, we could use this training set to train a perceptron or SVM. By doing so, we place the right weights on the opinion of each of the trees, in order to combine their opinions optimally.

12.5.9 Exercises for Section 12.5

Exercise 12.5.1: Suppose a training set has examples in four classes, and the fractions of examples in these classes are 1/2, 1/3, 1/8, and 1/24. What is the impurity of the root of a decision tree designed for this training set, if the measure of impurity is (a) Accuracy (b) GINI (c) Entropy?

Exercise 12.5.2: If a dataset consists of examples belonging to n different classes, what is the maximum impurity if the measure of impurity is (a) Accu-

racy (b) GINI (c) Entropy?

! **Exercise 12.5.3:** An important property of a function f is *convexity*, meaning that if $x < z < y$, then

$$f(z) > \frac{z-x}{y-x}f(x) + \frac{y-z}{y-x}f(y)$$

Less formally, the curve of f between x and y lies above the straight line between the points $\big(x, f(x)\big)$ and $\big(y, f(y)\big)$. In the following, assume there are two classes, and $f(x)$ is the impurity when x is the fraction of examples in the first class.

(a) Prove that the GINI impurity is convex.

(b) Prove that the Entropy measure of impurity is convex.

(c) Give an example to show that the Accuracy measure of impurity is not always convex. Hint: Note that convexity requires strict inequality; a straight line is not convex.

Exercise 12.5.4: To see why convexity is important, repeat the calculation of Fig. 12.27, but using Accuracy as the impurity measure. What goes wrong?

Exercise 12.5.5: Continuing Example 12.19, suppose we have replaced the node labeled "Continent = NA" by a leaf labeled "Baseball." Which other interior nodes do you think can be replaced by leaves without a significant increase to the error rate when the tree is applied to all countries of the world?

12.6 Comparison of Learning Methods

Each of the methods discussed in this chapter and elsewhere has its advantages. In this closing section, we shall consider:

- Does the method deal with categorical features or only with numerical features?
- Does the method deal effectively with high-dimensional feature vectors?
- Is the model that the method constructs intuitively understandable?

Perceptrons and Support-Vector Machines: These methods can handle millions of features, but they only make sense if the features are numerical. They only are effective if there is a linear separator, or at least a hyperplane that approximately separates the classes. However, we can separate points by a nonlinear boundary if we first transform the points to make the separator be linear. The model is expressed by a vector, the normal to the separating

hyperplane. Since this vector is often of very high dimension, it can be very hard to interpret the model.

Nearest-Neighbor Classification and Regression: Here, the model is the training set itself, so we expect it to be intuitively understandable. The approach can deal with multidimensional data, although the larger the number of dimensions, the sparser the training set will be, and therefore the less likely it is that we shall find a training point very close to the point we need to classify. That is, the "curse of dimensionality" makes nearest-neighbor methods questionable in high dimensions. These methods are really only useful for numerical features, although one could allow categorical features with a small number of values. For instance, a binary categorical feature like {male, female} could have the values replaced by 0 and 1, so there was no distance in this dimension between individuals of the same gender and distance 1 between other pairs of individuals. However, three or more values cannot be assigned numbers that are equidistant. Finally, nearest-neighbor methods have many parameters to set, including the distance measure we use (e.g., cosine or Euclidean), the number of neighbors to choose, and the kernel function to use. Different choices result in different classification, and in many cases it is not obvious which choices yield the best results.

Decision Trees: Unlike the other methods discussed in this chapter, Decision trees are useful for both categorical and numerical features. The models produced are generally quite understandable, since each decision is represented by one node of the tree. However, this approach is most useful for low-dimension feature vectors. As discussed in Section 12.5.7, building decision trees with many levels often leads to overfitting. But if a decision tree has few levels, then it cannot even mention more than a small number of features. As a result, the best use of decision trees is often to create a decision forest of many, low-depth trees and combine their decision in some way.

12.7 Summary of Chapter 12

- ✦ *Training Sets*: A training set consists of a feature vector, each component of which is a feature, and a label indicating the class to which the object represented by the feature vector belongs. Features can be categorical – belonging to an enumerated list of values – or numerical.

- ✦ *Test Sets and Overfitting*: When training some classifier on a training set, it is useful to remove some of the training set and use the removed data as a test set. After producing a model or classifier without using the test set, we can run the classifier on the test set to see how well it does. If the classifier does not perform as well on the test set as on the training set used, then we have overfit the training set by conforming to peculiarities of the training-set data which is not present in the data as a whole.

- *Batch Versus On-Line Learning*: In batch learning, the training set is available at any time and can be used in repeated passes. On-line learning uses a stream of training examples, each of which can be used only once.

- *Perceptrons*: This machine-learning method assumes the training set has only two class labels, positive and negative. Perceptrons work when there is a hyperplane that separates the feature vectors of the positive examples from those of the negative examples. We converge to that hyperplane by adjusting our estimate of the hyperplane by a fraction – the learning rate – of the direction that is the average of the currently misclassified points.

- *The Winnow Algorithm*: This algorithm is a variant of the perceptron algorithm that requires components of the feature vectors to be 0 or 1. Training examples are examined in a round-robin fashion, and if the current classification of a training example is incorrect, the components of the estimated separator where the feature vector has 1 are adjusted up or down, in the direction that will make it more likely this training example is correctly classified in the next round.

- *Nonlinear Separators*: When the training points do not have a linear function that separates two classes, it may still be possible to use a perceptron to classify them. We must find a function we can use to transform the points so that in the transformed space, the separator is a hyperplane.

- *Support-Vector Machines*: The SVM improves upon perceptrons by finding a separating hyperplane that not only separates the positive and negative points, but does so in a way that maximizes the margin – the distance perpendicular to the hyperplane to the nearest points. The points that lie exactly at this minimum distance are the support vectors. Alternatively, the SVM can be designed to allow points that are too close to the hyperplane, or even on the wrong side of the hyperplane, but minimize the error due to such misplaced points.

- *Solving the SVM Equations*: We can set up a function of the vector that is normal to the hyperplane, the length of the vector (which determines the margin), and the penalty for points on the wrong side of the margins. The regularization parameter determines the relative importance of a wide margin and a small penalty. The equations can be solved by several methods, including gradient descent and quadratic programming.

- *Gradient Descent*: This is a method for minimizing a loss function that depends on many variables and a training example, by repeatedly finding the (derivative) of the loss function with respect to each variable when given a training example, and moving the value of each variable in the direction that lowers the loss. The change in variables can either be an accumulation of the changes suggested by each training example (batch gradient descent), the result of one training example (stochastic gradient

descent), or the result of using a small subset of the training examples (minibatch gradient descent).

- ✦ *Nearest-Neighbor Learning*: In this approach to machine learning, the entire training set is used as the model. For each ("query") point to be classified, we search for its k nearest neighbors in the training set. The classification of the query point is some function of the labels of these k neighbors. The simplest case is when $k = 1$, in which case we can take the label of the query point to be the label of the nearest neighbor.
- ✦ *Regression*: A common case of nearest-neighbor learning, called regression, occurs when the there is only one feature vector, and it, as well as the label, are real numbers; i.e., the data defines a real-valued function of one variable. To estimate the label, i.e., the value of the function, for an unlabeled data point, we can perform some computation involving the k nearest neighbors. Examples include averaging the neighbors or taking a weighted average, where the weight of a neighbor is some decreasing function of its distance from the point whose label we are trying to determine.
- ✦ *Decision Trees*: This learning method constructs a tree where each interior node holds a test about the input and sends us to one of its children depending on the outcome of the test. Each leaf gives a decision about the class to which the input belongs.
- ✦ *Impurity Measures*: To help design a decision tree, we need a measure of how pure, i.e., close to a single class, is the set of training examples that reach a particular node of the decision tree. Possible measures of impurity include Accuracy (fraction of training examples with the wrong class), GINI (1 minus the squares of the fractions of examples in each of the classes), and Entropy (sum of the fractions of training examples in each class times the logarithm of the inverse of that fraction).
- ✦ *Designing a Decision-Tree Node*: We must consider each possible feature to use for the test at a node, and we must break the set of training examples that reach the node in a way that minimizes the average impurity of its children. For a numerical feature, we can order the traning examples by the value of that feature and use a test that breaks this list in a way that minimizes average impurity. For a categorical feature we order the values of that feature by the fraction of training examples with that value that belong to one particular class, and break the list to minimize average impurity.

12.8 References for Chapter 12

The perceptron was introduced in [14]. [8] introduces the idea of maximizing the margin around the separating hyperplane. A well-known book on the subject is [12].

The Winnow algorithm is from [11]. Also see the analysis in [2].

Support-vector machines appeared in [7]. [6] and [5] are useful surveys. [10] talks about a more efficient algorithm for the case of sparse features (most components of the feature vectors are zero). The use of gradient-descent methods is found in [3, 4].

For information about high-dimensional index structures for nearest-neighbor learning, see Chapter 14 of [9].

The original work on construction of decision trees is [13]. [1] is a popular paper describing the methodology used in this chapter.

1. H. Blockeel and L. De Raedt, "Top-down induction of first-order logical decision trees," *Artificial intelligence* **101**:1–2 (1998), pp. 285–297.

2. A. Blum, "Empirical support for winnow and weighted-majority algorithms: results on a calendar scheduling domain," *Machine Learning* **26** (1997), pp. 5–23.

3. L. Bottou, "Large-scale machine learning with stochastic gradient descent," *Proc. 19th Intl. Conf. on Computational Statistics* (2010), pp. 177–187, Springer.

4. L. Bottou, "Stochastic gradient tricks, neural networks," in *Tricks of the Trade, Reloaded*, pp. 430–445, Edited by G. Montavon, G.B. Orr and K.-R. Mueller, Lecture Notes in Computer Science (LNCS 7700), Springer, 2012.

5. C.J.C. Burges, "A tutorial on support vector machines for pattern recognition," *Data Mining and Knowledge Discovery* **2** (1998), pp. 121–167.

6. N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*, Cambridge University Press, 2000.

7. C. Cortes and V.N. Vapnik, "Support-vector networks," *Machine Learning* **20** (1995), pp. 273–297.

8. Y. Freund and R.E. Schapire, "Large margin classification using the perceptron algorithm," *Machine Learning* **37** (1999), pp. 277–296.

9. H. Garcia-Molina, J.D. Ullman, and J. Widom, *Database Systems: the Complete Book*, Prentice Hall, Upper Saddle River NJ, 2009.

10. T. Joachims, "Training linear SVMs in linear time." *Proc. 12th ACM SIGKDD* (2006), pp. 217–226.

11. N. Littlestone, "Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm," *Machine Learning* **2** (1988), pp. 285–318.

12. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry* (2nd edition), MIT Press, Cambridge MA, 1972.

13. J. R. Quinlan, "Induction of decision trees," *Machine Learning* **1** (1986), pp. 81–106.

14. F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological Review* **65**:6 (1958), pp. 386–408.