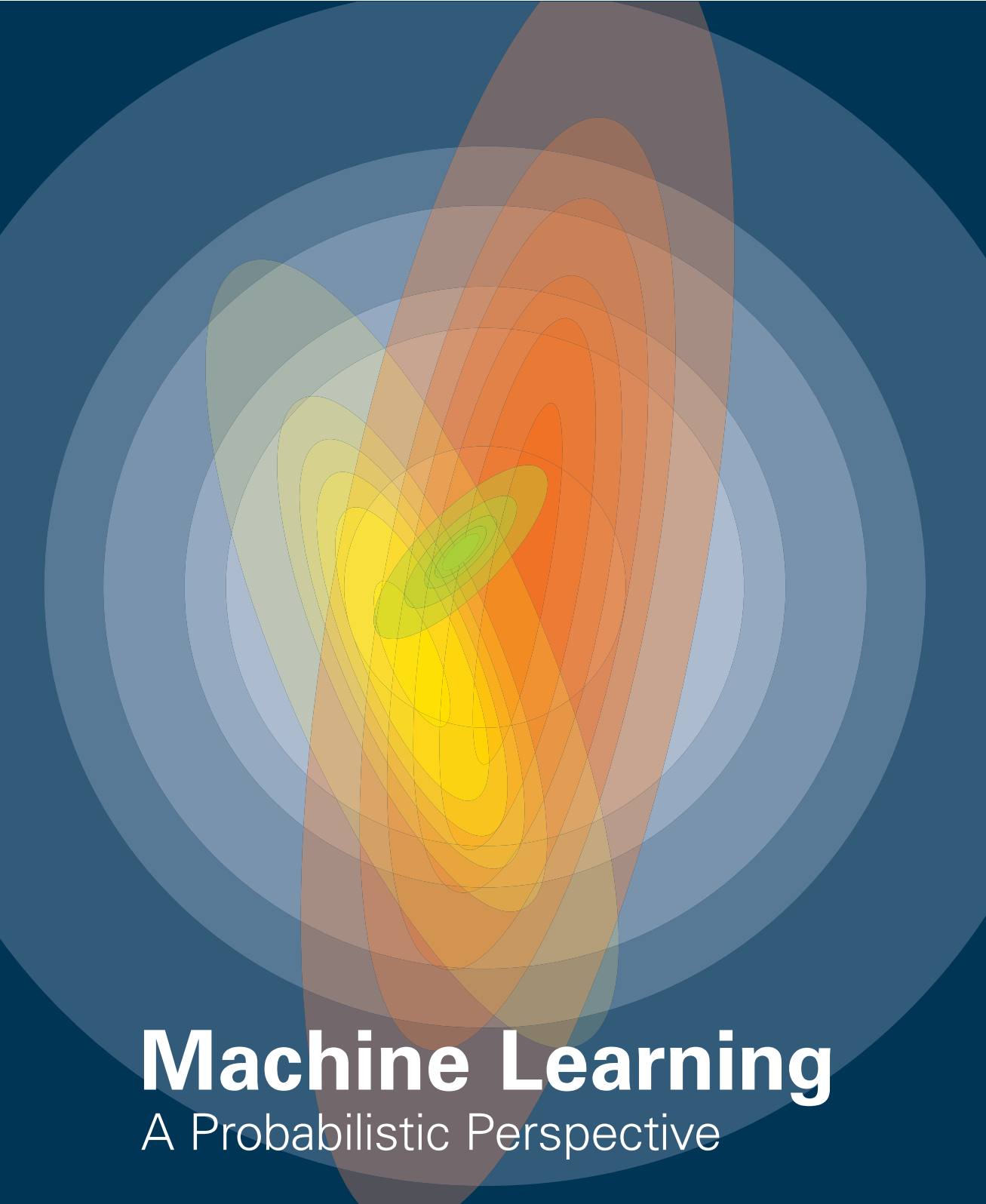




Machine Learning

A Probabilistic Perspective

Kevin P. Murphy

Machine Learning: A Probabilistic Perspective

Machine Learning

A Probabilistic Perspective

Kevin P. Murphy

The MIT Press
Cambridge, Massachusetts
London, England

© 2012 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

For information about special quantity discounts, please email special_sales@mitpress.mit.edu

This book was set in the $\text{\LaTeX}$ programming language by the author. Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Information

Murphy, Kevin P.
Machine learning : a probabilistic perspective / Kevin P. Murphy.
p. cm. — (Adaptive computation and machine learning series)
Includes bibliographical references and index.
ISBN 978-0-262-01802-9 (hardcover : alk. paper)
1. Machine learning. 2. Probabilities. I. Title.
Q325.5.M87 2012
006.3'1—dc23
2012004558

10 9 8 7 6 5 4 3 2 1

This book is dedicated to Alessandro, Michael and Stefano,
and to the memory of Gerard Joseph Murphy.

1 *Introduction*

1.1 Machine learning: what and why?

We are drowning in information and starving for knowledge. — John Naisbitt.

We are entering the era of **big data**. For example, there are about 1 trillion web pages¹; one hour of video is uploaded to YouTube every second, amounting to 10 years of content every day²; the genomes of 1000s of people, each of which has a length of 3.8×10^9 base pairs, have been sequenced by various labs; Walmart handles more than 1M transactions per hour and has databases containing more than 2.5 petabytes (2.5×10^{15}) of information (Cukier 2010); and so on.

This deluge of data calls for automated methods of data analysis, which is what **machine learning** provides. In particular, we define machine learning as a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty (such as planning how to collect more data!).

This book adopts the view that the best way to solve such problems is to use the tools of probability theory. Probability theory can be applied to any problem involving uncertainty. In machine learning, uncertainty comes in many forms: what is the best prediction about the future given some past data? what is the best model to explain some data? what measurement should I perform next? etc. The probabilistic approach to machine learning is closely related to the field of statistics, but differs slightly in terms of its emphasis and terminology³.

We will describe a wide variety of probabilistic models, suitable for a wide variety of data and tasks. We will also describe a wide variety of algorithms for learning and using such models. The goal is not to develop a cook book of ad hoc techniques, but instead to present a unified view of the field through the lens of probabilistic modeling and inference. Although we will pay attention to computational efficiency, details on how to scale these methods to truly massive datasets are better described in other books, such as (Rajaraman and Ullman 2011; Bekkerman et al. 2011).

1. <http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html>

2. Source: http://www.youtube.com/t/press_statistics.

3. Rob Tibshirani, a statistician at Stanford university, has created an amusing comparison between machine learning and statistics, available at <http://www-stat.stanford.edu/~tibs/stat315a/glossary.pdf>.

It should be noted, however, that even when one has an apparently massive data set, the effective number of data points for certain cases of interest might be quite small. In fact, data across a variety of domains exhibits a property known as the **long tail**, which means that a few things (e.g., words) are very common, but most things are quite rare (see Section 2.4.6 for details). For example, 20% of Google searches each day have never been seen before⁴. This means that the core statistical issues that we discuss in this book, concerning generalizing from relatively small samples sizes, are still very relevant even in the big data era.

1.1.1 Types of machine learning

Machine learning is usually divided into two main types. In the **predictive** or **supervised learning** approach, the goal is to learn a mapping from inputs $\mathbf{x}$ to outputs y , given a labeled set of input-output pairs $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$. Here $\mathcal{D}$ is called the **training set**, and N is the number of training examples.

In the simplest setting, each training input $\mathbf{x}_i$ is a D -dimensional vector of numbers, representing, say, the height and weight of a person. These are called **features**, **attributes** or **covariates**. In general, however, $\mathbf{x}_i$ could be a complex structured object, such as an image, a sentence, an email message, a time series, a molecular shape, a graph, etc.

Similarly the form of the output or **response variable** can in principle be anything, but most methods assume that y_i is a **categorical** or **nominal** variable from some finite set, $y_i \in \{1, \dots, C\}$ (such as male or female), or that y_i is a real-valued scalar (such as income level). When y_i is categorical, the problem is known as **classification** or **pattern recognition**, and when y_i is real-valued, the problem is known as **regression**. Another variant, known as **ordinal regression**, occurs where label space $\mathcal{Y}$ has some natural ordering, such as grades A–F.

The second main type of machine learning is the **descriptive** or **unsupervised learning** approach. Here we are only given inputs, $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^N$, and the goal is to find “interesting patterns” in the data. This is sometimes called **knowledge discovery**. This is a much less well-defined problem, since we are not told what kinds of patterns to look for, and there is no obvious error metric to use (unlike supervised learning, where we can compare our prediction of y for a given $\mathbf{x}$ to the observed value).

There is a third type of machine learning, known as **reinforcement learning**, which is somewhat less commonly used. This is useful for learning how to act or behave when given occasional reward or punishment signals. (For example, consider how a baby learns to walk.) Unfortunately, RL is beyond the scope of this book, although we do discuss decision theory in Section 5.7, which is the basis of RL. See e.g., (Kaelbling et al. 1996; Sutton and Barto 1998; Russell and Norvig 2010; Szepesvari 2010; Wiering and van Otterlo 2012) for more information on RL.

4.

<http://certifiedknowledge.org/blog/are-search-queries-becoming-even-more-unique-statistics-from-google>.

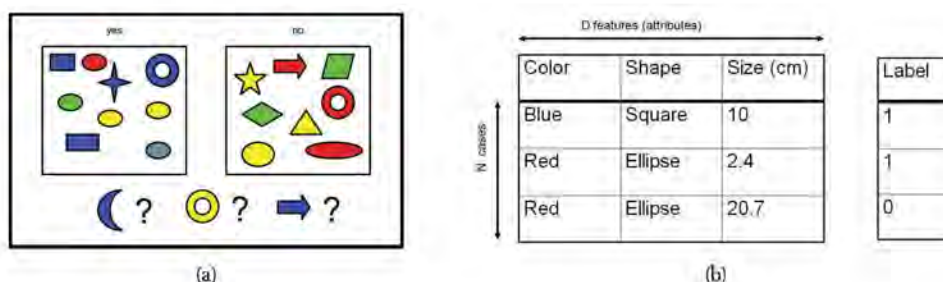


Figure 1.1 Left: Some labeled training examples of colored shapes, along with 3 unlabeled test cases. Right: Representing the training data as an $N \times D$ design matrix. Row i represents the feature vector $\mathbf{x}_i$. The last column is the label, $y_i \in \{0, 1\}$. Based on a figure by Leslie Kaelbling.

1.2 Supervised learning

We begin our investigation of machine learning by discussing supervised learning, which is the form of ML most widely used in practice.

1.2.1 Classification

In this section, we discuss classification. Here the goal is to learn a mapping from inputs $\mathbf{x}$ to outputs y , where $y \in \{1, \dots, C\}$, with C being the number of classes. If $C = 2$, this is called **binary classification** (in which case we often assume $y \in \{0, 1\}$); if $C > 2$, this is called **multiclass classification**. If the class labels are not mutually exclusive (e.g., somebody may be classified as tall and strong), we call it **multi-label classification**, but this is best viewed as predicting multiple related binary class labels (a so-called **multiple output model**). When we use the term “classification”, we will mean multiclass classification with a single output, unless we state otherwise.

One way to formalize the problem is as **function approximation**. We assume $y = f(\mathbf{x})$ for some unknown function f , and the goal of learning is to estimate the function f given a labeled training set, and then to make predictions using $\hat{y} = \hat{f}(\mathbf{x})$. (We use the hat symbol to denote an estimate.) Our main goal is to make predictions on novel inputs, meaning ones that we have not seen before (this is called **generalization**), since predicting the response on the training set is easy (we can just look up the answer).

1.2.1.1 Example

As a simple toy example of classification, consider the problem illustrated in Figure 1.1(a). We have two classes of object which correspond to labels 0 and 1. The inputs are colored shapes. These have been described by a set of D features or attributes, which are stored in an $N \times D$ design matrix $\mathbf{X}$, shown in Figure 1.1(b). The input features $\mathbf{x}$ can be discrete, continuous or a combination of the two. In addition to the inputs, we have a vector of training labels $\mathbf{y}$.

In Figure 1.1, the test cases are a blue crescent, a yellow circle and a blue arrow. None of these have been seen before. Thus we are required to **generalize** beyond the training set. A

reasonable guess is that blue crescent should be $y = 1$, since all blue shapes are labeled 1 in the training set. The yellow circle is harder to classify, since some yellow things are labeled $y = 1$ and some are labeled $y = 0$, and some circles are labeled $y = 1$ and some $y = 0$. Consequently it is not clear what the right label should be in the case of the yellow circle. Similarly, the correct label for the blue arrow is unclear.

1.2.1.2 The need for probabilistic predictions

To handle ambiguous cases, such as the yellow circle above, it is desirable to return a probability. The reader is assumed to already have some familiarity with basic concepts in probability. If not, please consult Chapter 2 for a refresher, if necessary.

We will denote the probability distribution over possible labels, given the input vector $\mathbf{x}$ and training set $\mathcal{D}$ by $p(y|\mathbf{x}, \mathcal{D})$. In general, this represents a vector of length C . (If there are just two classes, it is sufficient to return the single number $p(y = 1|\mathbf{x}, \mathcal{D})$, since $p(y = 1|\mathbf{x}, \mathcal{D}) + p(y = 0|\mathbf{x}, \mathcal{D}) = 1$.) In our notation, we make explicit that the probability is conditional on the test input $\mathbf{x}$, as well as the training set $\mathcal{D}$, by putting these terms on the right hand side of the conditioning bar $|$. We are also implicitly conditioning on the form of model that we use to make predictions. When choosing between different models, we will make this assumption explicit by writing $p(y|\mathbf{x}, \mathcal{D}, M)$, where M denotes the model. However, if the model is clear from context, we will drop M from our notation for brevity.

Given a probabilistic output, we can always compute our “best guess” as to the “true label” using

$$\hat{y} = \hat{f}(\mathbf{x}) = \arg\max_{c=1}^C p(y = c|\mathbf{x}, \mathcal{D}) \quad (1.1)$$

This corresponds to the most probable class label, and is called the **mode** of the distribution $p(y|\mathbf{x}, \mathcal{D})$; it is also known as a **MAP estimate** (MAP stands for **maximum a posteriori**). Using the most probable label makes intuitive sense, but we will give a more formal justification for this procedure in Section 5.7.

Now consider a case such as the yellow circle, where $p(\hat{y}|\mathbf{x}, \mathcal{D})$ is far from 1.0. In such a case we are not very confident of our answer, so it might be better to say “I don’t know” instead of returning an answer that we don’t really trust. This is particularly important in domains such as medicine and finance where we may be risk averse, as we explain in Section 5.7. Another application where it is important to assess risk is when playing TV game shows, such as Jeopardy. In this game, contestants have to solve various word puzzles and answer a variety of trivia questions, but if they answer incorrectly, they lose money. In 2011, IBM unveiled a computer system called Watson which beat the top human Jeopardy champion. Watson uses a variety of interesting techniques (Ferrucci et al. 2010), but the most pertinent one for our present purposes is that it contains a module that estimates how confident it is of its answer. The system only chooses to “buzz in” its answer if sufficiently confident it is correct. Similarly, Google has a system known as SmartASS (ad selection system) that predicts the probability you will click on an ad based on your search history and other user and ad-specific features (Metz 2010). This probability is known as the **click-through rate** or **CTR**, and can be used to maximize expected profit. We will discuss some of the basic principles behind systems such as SmartASS later in this book.

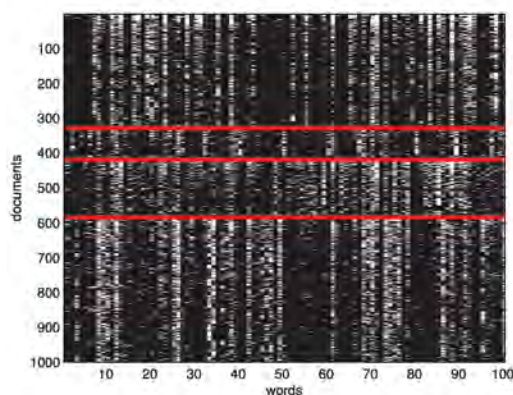


Figure 1.2 Subset of size 16242 x 100 of the 20-newsgroups data. We only show 1000 rows, for clarity. Each row is a document (represented as a bag-of-words bit vector), each column is a word. The red lines separate the 4 classes, which are (in descending order) comp, rec, sci, talk (these are the titles of USENET groups). We can see that there are subsets of words whose presence or absence is indicative of the class. The data is available from <http://cs.nyu.edu/~roweis/data.html>. Figure generated by newsgroupsVisualize.

1.2.1.3 Real-world applications

Classification is probably the most widely used form of machine learning, and has been used to solve many interesting and often difficult real-world problems. We have already mentioned some important applications. We give a few more examples below.

Document classification and email spam filtering

In **document classification**, the goal is to classify a document, such as a web page or email message, into one of C classes, that is, to compute $p(y = c | \mathbf{x}, \mathcal{D})$, where $\mathbf{x}$ is some representation of the text. A special case of this is **email spam filtering**, where the classes are spam $y = 1$ or ham $y = 0$.

Most classifiers assume that the input vector $\mathbf{x}$ has a fixed size. A common way to represent variable-length documents in feature-vector format is to use a **bag of words** representation. This is explained in detail in Section 3.4.4.1, but the basic idea is to define $x_{ij} = 1$ iff word j occurs in document i . If we apply this transformation to every document in our data set, we get a binary document $\times$ word co-occurrence matrix: see Figure 1.2 for an example. Essentially the document classification problem has been reduced to one that looks for subtle changes in the pattern of bits. For example, we may notice that most spam messages have a high probability of containing the words “buy”, “cheap”, “viagra”, etc. In Exercise 8.1 and Exercise 8.2, you will get hands-on experience applying various classification techniques to the spam filtering problem.

where $\phi_m(\mathbf{x})$ is the m 'th basis function, which is learned from data. This framework covers all of the models we will discuss in this chapter.

Typically the basis functions are parametric, so we can write $\phi_m(\mathbf{x}) = \phi(\mathbf{x}; \mathbf{v}_m)$, where $\mathbf{v}_m$ are the parameters of the basis function itself. We will use $\boldsymbol{\theta} = (w_0, \mathbf{w}_{1:M}, \{\mathbf{v}_m\}_{m=1}^M)$ to denote the entire parameter set. The resulting model is not linear-in-the-parameters anymore, so we will only be able to compute a locally optimal MLE or MAP estimate of $\boldsymbol{\theta}$. Nevertheless, such models often significantly outperform linear models, as we will see.

16.2 Classification and regression trees (CART)

Classification and regression trees or **CART** models, also called **decision trees** (not to be confused with the decision trees used in decision theory) are defined by recursively partitioning the input space, and defining a local model in each resulting region of input space. This can be represented by a tree, with one leaf per region, as we explain below.

16.2.1 Basics

To explain the CART approach, consider the tree in Figure 16.1(a). The first node asks if x_1 is less than some threshold t_1 . If yes, we then ask if x_2 is less than some other threshold t_2 . If yes, we are in the bottom left quadrant of space, R_1 . If no, we ask if x_1 is less than t_3 . And so on. The result of these **axis parallel splits** is to partition 2d space into 5 regions, as shown in Figure 16.1(b). We can now associate a mean response with each of these regions, resulting in the piecewise constant surface shown in Figure 16.1(c).

We can write the model in the following form

$$f(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}] = \sum_{m=1}^M w_m \mathbb{I}(\mathbf{x} \in R_m) = \sum_{m=1}^M w_m \phi(\mathbf{x}; \mathbf{v}_m) \quad (16.4)$$

where R_m is the m 'th region, w_m is the mean response in this region, and $\mathbf{v}_m$ encodes the choice of variable to split on, and the threshold value, on the path from the root to the m 'th leaf. This makes it clear that a CART model is just an adaptive basis-function model, where the basis functions define the regions, and the weights specify the response value in each region. We discuss how to find these basis functions below.

We can generalize this to the classification setting by storing the distribution over class labels in each leaf, instead of the mean response. This is illustrated in Figure 16.2. This model can be used to classify the data in Figure 1.1. For example, we first check the color of the object. If it is blue, we follow the left branch and end up in a leaf labeled “4,0”, which means we have 4 positive examples and 0 negative examples which match this criterion. Hence we predict $p(y = 1|\mathbf{x}) = 4/4$ if $\mathbf{x}$ is blue. If it is red, we then check the shape: if it is an ellipse, we end up in a leaf labeled “1,1”, so we predict $p(y = 1|\mathbf{x}) = 1/2$. If it is red but not an ellipse, we predict $p(y = 1|\mathbf{x}) = 0/2$; If it is some other colour, we check the size: if less than 10, we predict $p(y = 1|\mathbf{x}) = 4/4$, otherwise $p(y = 1|\mathbf{x}) = 0/5$. These probabilities are just the empirical fraction of positive examples that satisfy each conjunction of feature values, which defines a path from the root to a leaf.

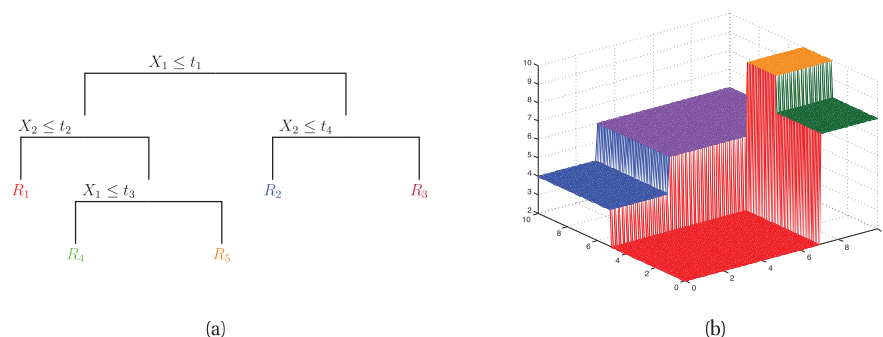


Figure 16.1 A simple regression tree on two inputs. Based on Figure 9.2 of (Hastie et al. 2009). Figure generated by `regtreeSurfaceDemo`.

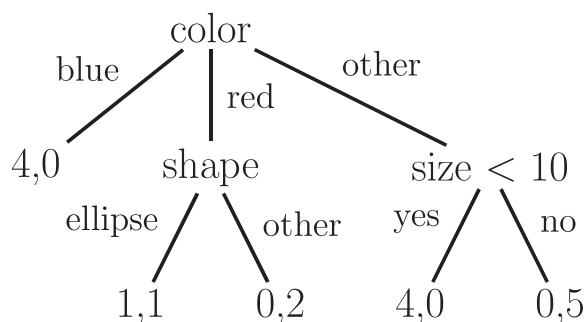


Figure 16.2 A simple decision tree for the data in Figure 1.1. A leaf labeled as (n_1, n_0) means that there are n_1 positive examples that match this path, and n_0 negative examples. In this tree, most of the leaves are “pure”, meaning they only have examples of one class or the other; the only exception is leaf representing red ellipses, which has a label distribution of $(1, 1)$. We could distinguish positive from negative red ellipses by adding a further test based on size. However, it is not always desirable to construct trees that perfectly model the training data, due to overfitting.

16.2.2 Growing a tree

Finding the optimal partitioning of the data is NP-complete (Hyafil and Rivest 1976), so it is common to use the greedy procedure shown in Algorithm 6 to compute a locally optimal MLE. This method is used by CART, (Breiman et al. 1984) **C4.5**(Quinlan 1993), and **ID3** (Quinlan 1986), which are three popular implementations of the method. (See `dtfit` for a simple Matlab implementation.)

The split function chooses the best feature, and the best value for that feature, as follows:

$$(j^*, t^*) = \arg \min_{j \in \{1, \dots, D\}} \min_{t \in \mathcal{T}_j} \text{cost}(\{\mathbf{x}_i, y_i : x_{ij} \leq t\}) + \text{cost}(\{\mathbf{x}_i, y_i : x_{ij} > t\}) \quad (16.5)$$

Algorithm 16.1: Recursive procedure to grow a classification/ regression tree

```

1 function fitTree(node,  $\mathcal{D}$ , depth) ;
2   node.prediction = mean( $y_i : i \in \mathcal{D}$ ) // or class label distribution ;
3   ( $j^*, t^*, \mathcal{D}_L, \mathcal{D}_R$ ) = split( $\mathcal{D}$ );
4   if not worthSplitting(depth, cost,  $\mathcal{D}_L, \mathcal{D}_R$ ) then
5     return node
6   else
7     node.test =  $\lambda \mathbf{x}.x_{j^*} < t^*$  // anonymous function;
8     node.left = fitTree(node,  $\mathcal{D}_L$ , depth+1);
9     node.right = fitTree(node,  $\mathcal{D}_R$ , depth+1);
10    return node;
```

where the cost function for a given dataset will be defined below. For notational simplicity, we have assumed all inputs are real-valued or ordinal, so it makes sense to compare a feature x_{ij} to a numeric value t . The set of possible thresholds $\mathcal{T}_j$ for feature j can be obtained by sorting the unique values of x_{ij} . For example, if feature 1 has the values $\{4.5, -12, 72, -12\}$, then we set $\mathcal{T}_1 = \{-12, 4.5, 72\}$. In the case of categorical inputs, the most common approach is to consider splits of the form $x_{ij} = c_k$ and $x_{ij} \neq c_k$, for each possible class label c_k . Although we could allow for multi-way splits (resulting in non-binary trees), this would result in **data fragmentation**, meaning too little data might “fall” into each subtree, resulting in overfitting.

The function that checks if a node is worth splitting can use several stopping heuristics, such as the following:

- is the reduction in cost too small? Typically we define the gain of using a feature to be a normalized measure of the reduction in cost:

$$\Delta \triangleq \text{cost}(\mathcal{D}) - \left(\frac{|\mathcal{D}_L|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_L) + \frac{|\mathcal{D}_R|}{|\mathcal{D}|} \text{cost}(\mathcal{D}_R) \right) \quad (16.6)$$

- has the tree exceeded the maximum desired depth?
- is the distribution of the response in either $\mathcal{D}_L$ or $\mathcal{D}_R$ sufficiently homogeneous (e.g., all labels are the same, so the distribution is **pure**)?
- is the number of examples in either $\mathcal{D}_L$ or $\mathcal{D}_R$ too small?

All that remains is to specify the cost measure used to evaluate the quality of a proposed split. This depends on whether our goal is regression or classification. We discuss both cases below.

16.2.2.1 Regression cost

In the regression setting, we define the cost as follows:

$$\text{cost}(\mathcal{D}) = \sum_{i \in \mathcal{D}} (y_i - \bar{y})^2 \quad (16.7)$$

where $\bar{y} = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} y_i$ is the mean of the response variable in the specified set of data. Alternatively, we can fit a linear regression model for each leaf, using as inputs the features that were chosen on the path from the root, and then measure the residual error.

16.2.2.2 Classification cost

In the classification setting, there are several ways to measure the quality of a split. First, we fit a multinoulli model to the data in the leaf satisfying the test $X_j < t$ by estimating the class-conditional probabilities as follows:

$$\hat{\pi}_c = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}(y_i = c) \quad (16.8)$$

where $\mathcal{D}$ is the data in the leaf. Given this, there are several common error measures for evaluating a proposed partition:

- **Misclassification rate.** We define the most probable class label as $\hat{y}_c = \operatorname{argmax}_c \hat{\pi}_c$. The corresponding error rate is then

$$\frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}(y_i \neq \hat{y}) = 1 - \hat{\pi}_{\hat{y}} \quad (16.9)$$

- **Entropy, or deviance:**

$$\mathbb{H}(\hat{\pi}) = - \sum_{c=1}^C \hat{\pi}_c \log \hat{\pi}_c \quad (16.10)$$

Note that minimizing the entropy is equivalent to maximizing the **information gain** (Quinlan 1986) between test $X_j < t$ and the class label Y , defined by

$$\text{infoGain}(X_j < t, Y) \triangleq \mathbb{H}(Y) - \mathbb{H}(Y|X_j < t) \quad (16.11)$$

$$= \left(- \sum_c p(y = c) \log p(y = c) \right) \quad (16.12)$$

$$+ \left(\sum_c p(y = c|X_j < t) \log p(c|X_j < t) \right) \quad (16.13)$$

since $\hat{\pi}_c$ is an MLE for the distribution $p(c|X_j < t)$.¹

1. If X_j is categorical, and we use tests of the form $X_j = k$, then taking expectations over values of X_j gives the mutual information between X_j and Y : $\mathbb{E}[\text{infoGain}(X_j, Y)] = \sum_k p(X_j = k) \text{infoGain}(X_j = k, Y) = \mathbb{H}(Y) - \mathbb{H}(Y|X_j) = \mathbb{I}(Y; X_j)$.

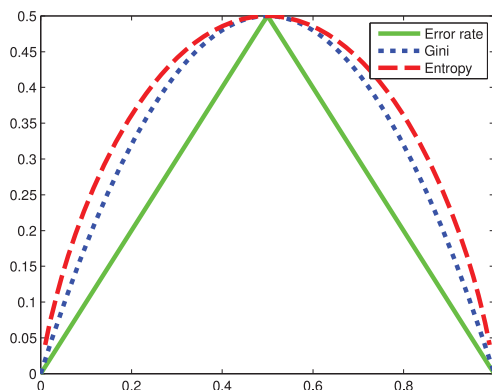


Figure 16.3 Node impurity measures for binary classification. The horizontal axis corresponds to p , the probability of class 1. The entropy measure has been rescaled to pass through $(0.5, 0.5)$. Based on Figure 9.3 of (Hastie et al. 2009). Figure generated by `giniDemo`.

- **Gini index**

$$\sum_{c=1}^C \hat{\pi}_c(1 - \hat{\pi}_c) = \sum_c \hat{\pi}_c - \sum_c \hat{\pi}_c^2 = 1 - \sum_c \hat{\pi}_c^2 \quad (16.14)$$

This is the expected error rate. To see this, note that $\hat{\pi}_c$ is the probability a random entry in the leaf belongs to class c , and $(1 - \hat{\pi}_c)$ is the probability it would be misclassified.

In the two-class case, where $p = \pi_m(1)$, the misclassification rate is $1 - \max(p, 1 - p)$, the entropy is $\mathbb{H}_2(p)$, and the Gini index is $2p(1 - p)$. These are plotted in Figure 16.3. We see that the cross-entropy and Gini measures are very similar, and are more sensitive to changes in class probability than is the misclassification rate. For example, consider a two-class problem with 400 cases in each class. Suppose one split created the nodes (300,100) and (100,300), while the other created the nodes (200,400) and (200,0). Both splits produce a misclassification rate of 0.25. However, the latter seems preferable, since one of the nodes is **pure**, i.e., it only contains one class. The cross-entropy and Gini measures will favor this latter choice.

16.2.2.3 Example

As an example, consider two of the four features from the 3-class iris dataset, shown in Figure 16.4(a). The resulting tree is shown in Figure 16.5(a), and the decision boundaries are shown in Figure 16.4(b). We see that the tree is quite complex, as are the resulting decision boundaries. In Figure 16.5(b), we show that the CV estimate of the error is much higher than the training set error, indicating overfitting. Below we discuss how to perform a tree-pruning stage to simplify the tree.

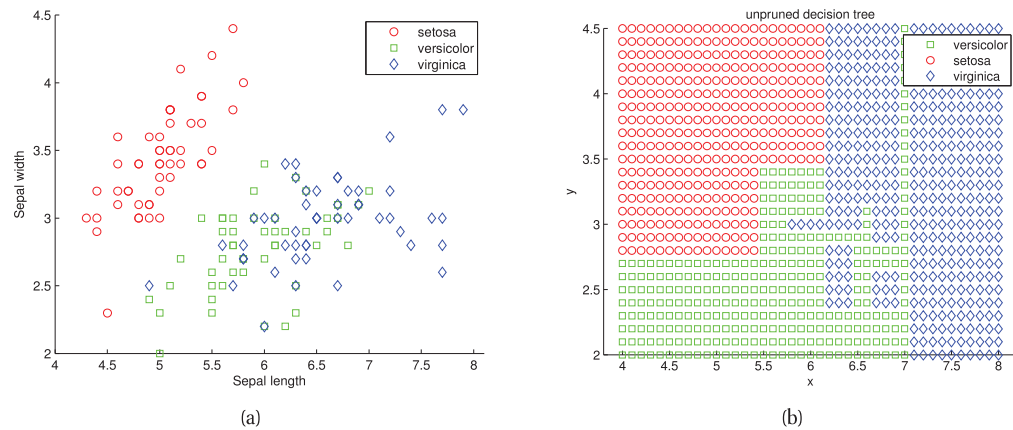


Figure 16.4 (a) Iris data. We only show the first two features, sepal length and sepal width, and ignore petal length and petal width. (b) Decision boundaries induced by the decision tree in Figure 16.5(a).

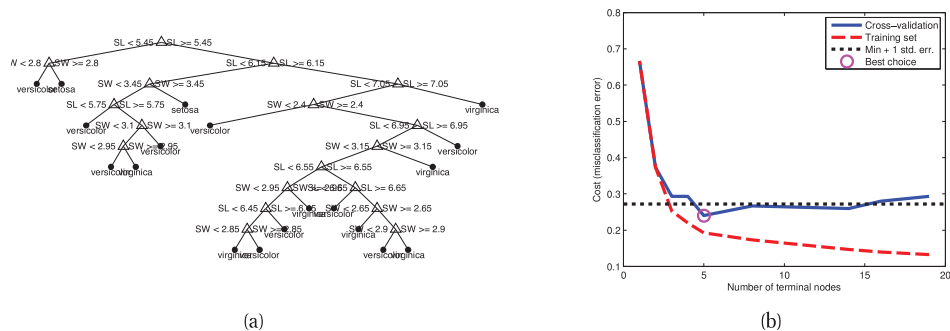


Figure 16.5 (a) Unpruned decision tree for Iris data. (b) Plot of misclassification error rate vs depth of tree. Figure generated by `dtreeDemoIris`.

16.2.3 Pruning a tree

To prevent overfitting, we can stop growing the tree if the decrease in the error is not sufficient to justify the extra complexity of adding an extra subtree. However, this tends to be too myopic. For example, on the xor data in Figure 14.2(c), it would might never make any splits, since each feature on its own has little predictive power.

The standard approach is therefore to grow a “full” tree, and then to perform **pruning**. This can be done using a scheme that prunes the branches giving the least increase in the error. See (Breiman et al. 1984) for details.

To determine how far to prune back, we can evaluate the cross-validated error on each such subtree, and then pick the tree whose CV error is within 1 standard error of the minimum. This is illustrated in Figure 16.4(b). The point with the minimum CV error corresponds to the simple tree in Figure 16.6(a).

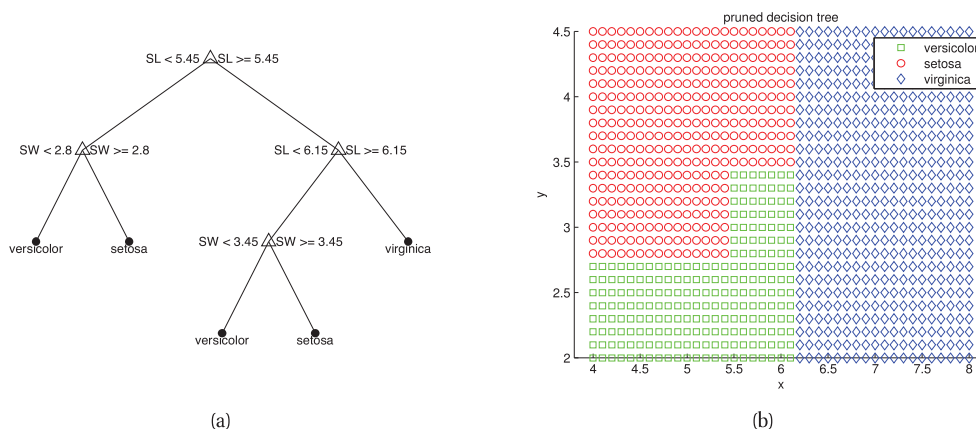


Figure 16.6 Pruned decision tree for Iris data. Figure generated by `dtreeDemoIris`.

16.2.4 Pros and cons of trees

CART models are popular for several reasons: they are easy to interpret², they can easily handle mixed discrete and continuous inputs, they are insensitive to monotone transformations of the inputs (because the split points are based on ranking the data points), they perform automatic variable selection, they are relatively robust to outliers, they scale well to large data sets, and they can be modified to handle missing inputs.³

However, CART models also have some disadvantages. The primary one is that they do not predict very accurately compared to other kinds of model. This is in part due to the greedy nature of the tree construction algorithm. A related problem is that trees are **unstable**: small changes to the input data can have large effects on the structure of the tree, due to the hierarchical nature of the tree-growing process, causing errors at the top to affect the rest of the tree. In frequentist terminology, we say that trees are high variance estimators. We discuss a solution to this below.

16.2.5 Random forests

One way to reduce the variance of an estimate is to average together many estimates. For example, we can train M different trees on different subsets of the data, chosen randomly with

2. We can postprocess the tree to derive a series of logical **rules** such as “If $x_1 < 5.45$ then ...” (Quinlan 1990).

3. The standard heuristic for handling missing inputs in decision trees is to look for a series of “backup” variables, which can induce a similar partition to the chosen variable at any given split; these can be used in case the chosen variable is unobserved at test time. These are called **surrogate splits**. This method finds highly correlated features, and can be thought of as learning a local joint model of the input. This has the advantage over a generative model of not modeling the entire joint distribution of inputs, but it has the disadvantage of being entirely ad hoc. A simpler approach, applicable to categorical variables, is to code “missing” as a new value, and then to treat the data as fully observed.

replacement, and then compute the ensemble

$$f(\mathbf{x}) = \sum_{m=1}^M \frac{1}{M} f_m(\mathbf{x}) \quad (16.15)$$

where f_m is the m 'th tree. This technique is called **bagging** (Breiman 1996), which stands for “bootstrap aggregating”.

Unfortunately, simply re-running the same learning algorithm on different subsets of the data can result in highly correlated predictors, which limits the amount of variance reduction that is possible. The technique known as **random forests** (Breiman 2001a) tries to decorrelate the base learners by learning trees based on a randomly chosen subset of input variables, as well as a randomly chosen subset of data cases. Such models often have very good predictive accuracy (Caruana and Niculescu-Mizil 2006), and have been widely used in many applications (e.g., for body pose recognition using Microsoft's popular kinect sensor (Shotton et al. 2011)).

Bagging is a frequentist concept. It is also possible to adopt a Bayesian approach to learning trees. In particular, (Chipman et al. 1998; Denison et al. 1998; Wu et al. 2007) perform approximate inference over the space of trees (structure and parameters) using MCMC. This reduces the variance of the predictions. We can also perform Bayesian inference over the space of ensembles of trees, which tends to work much better. This is known as **Bayesian adaptive regression trees** or **BART** (Chipman et al. 2010). Note that the cost of these sampling-based Bayesian methods is comparable to the sampling-based random forest method. That is, both approaches are fairly slow to train, but produce high quality classifiers.

Unfortunately, methods that use multiple trees (whether derived from a Bayesian or frequentist standpoint) lose their nice interpretability properties. Fortunately, various post-processing measures can be applied, as discussed in Section 16.8.

16.2.6 CART compared to hierarchical mixture of experts *

An interesting alternative to a decision tree is known as the hierarchical mixture of experts. Figure 11.7(b) gives an illustration where we have two levels of experts. This can be thought of as a probabilistic decision tree of depth 2, since we recursively partition the space, and apply a different expert to each partition. Hastie et al. (Hastie et al. 2009, p331) write that “The HME approach is a promising competitor to CART trees”. Some of the advantages include the following:

- The model can partition the input space using any set of nested linear decision boundaries. By contrast, standard decision trees are constrained to use axis-parallel splits.
- The model makes predictions by averaging over all experts. By contrast, in a standard decision tree, predictions are made only based on the model in the corresponding leaf. Since leaves often contain few training examples, this can result in overfitting.
- Fitting an HME involves solving a smooth continuous optimization problem (usually using EM), which is likely to be less prone to local optima than the standard greedy discrete optimization methods used to fit decision trees. For similar reasons, it is computationally easier to “be Bayesian” about the parameters of an HME (see e.g., (Peng et al. 1996; Bishop

and Svensén 2003)) than about the structure and parameters of a decision tree (see e.g., (Wu et al. 2007)).

16.3 Generalized additive models

A simple way to create a nonlinear model with multiple inputs is to use a **generalized additive model** (Hastie and Tibshirani 1990), which is a model of the form

$$f(\mathbf{x}) = \alpha + f_1(x_1) + \cdots + f_D(x_D) \quad (16.16)$$

Here each f_j can be modeled by some scatterplot smoother, and $f(\mathbf{x})$ can be mapped to $p(y|\mathbf{x})$ using a link function, as in a GLM (hence the term *generalized* additive model).

If we use regression splines (or some other fixed basis function expansion approach) for the f_j , then each $f_j(x_j)$ can be written as $\beta_j^T \phi_j(x_j)$, so the whole model can be written as $f(\mathbf{x}) = \beta^T \phi(\mathbf{x})$, where $\phi(\mathbf{x}) = [1, \phi_1(x_1), \dots, \phi_D(x_D)]$. However, it is more common to use smoothing splines (Section 15.4.6) for the f_j . In this case, the objective (in the regression setting) becomes

$$J(\alpha, f_1, \dots, f_D) = \sum_{i=1}^N \left(y_i - \alpha - \sum_{j=1}^D f_j(x_{ij}) \right)^2 + \sum_{j=1}^D \lambda_j \int f_j''(t_j)^2 dt_j \quad (16.17)$$

where λ_j is the strength of the regularizer for f_j .

16.3.1 Backfitting

We now discuss how to fit the model using MLE. The constant α is not uniquely identifiable, since we can always add or subtract constants to any of the f_j functions. The convention is to assume $\sum_{i=1}^N f_j(x_{ij}) = 0$ for all j . In this case, the MLE for α is just $\hat{\alpha} = \frac{1}{N} \sum_{i=1}^N y_i$.

To fit the rest of the model, we can center the responses (by subtracting $\hat{\alpha}$), and then iteratively update each f_j in turn, using as a target vector the residuals obtained by omitting term f_j :

$$\hat{f}_j := \text{smoother}(\{y_i - \sum_{k \neq j} \hat{f}_k(x_{ik})\}_{i=1}^N) \quad (16.18)$$

We should then ensure the output is zero mean using

$$\hat{f}_j := \hat{f}_j - \frac{1}{N} \sum_{i=1}^N \hat{f}_j(x_{ij}) \quad (16.19)$$

This is called the **backfitting** algorithm (Hastie and Tibshirani 1990). If $\mathbf{X}$ has full column rank, then the above objective is convex (since each smoothing spline is a linear operator, as shown in Section 15.4.2), so this procedure is guaranteed to converge to the global optimum.

In the GLM case, we need to modify the method somewhat. The basic idea is to replace the weighted least squares step of IRLS (see Section 8.3.4) with a weighted backfitting algorithm. In the logistic regression case, each response has weight $s_i = \mu_i(1 - \mu_i)$ associated with it, where $\mu_i = \text{sigm}(\hat{\alpha} + \sum_{j=1}^D \hat{f}_j(x_{ij}))$.

16.3.2 Computational efficiency

Each call to the smoother takes $O(N)$ time, so the total cost is $O(NDT)$, where T is the number of iterations. If we have high-dimensional inputs, fitting a GAM is expensive. One approach is to combine it with a sparsity penalty, see e.g., the **SpAM** (sparse additive model) approach of (Ravikumar et al. 2009). Alternatively, we can use a greedy approach, such as boosting (see Section 16.4.6)

16.3.3 Multivariate adaptive regression splines (MARS)

We can extend GAMs by allowing for interaction effects. In general, we can create an ANOVA decomposition:

$$f(\mathbf{x}) = \beta_0 + \sum_{j=1}^D f_j(x_j) + \sum_{j,k} f_{jk}(x_j, x_k) + \sum_{j,k,l} f_{jkl}(x_j, x_k, x_l) + \cdots \quad (16.20)$$

Of course, we cannot allow for too many higher-order interactions, because there will be too many parameters to fit.

It is common to use greedy search to decide which variables to add. The **multivariate adaptive regression splines** or **MARS** algorithm is one example of this (Hastie et al. 2009, Sec9.4). It fits models of the form in Equation 16.20, where it uses a tensor product basis of regression splines to represent the multidimensional regression functions. For example, for 2d input, we might use

$$\begin{aligned} f(x_1, x_2) = & \beta_0 + \sum_m \beta_{1m}(x_1 - t_{1m})_+ \\ & + \sum_m \beta_{2m}(t_{2m} - x_2)_+ + \sum_m \beta_{12m}(x_1 - t_{1m})_+(t_{2m} - x_2)_+ \end{aligned} \quad (16.21)$$

To create such a function, we start with a set of candidate basis functions of the form

$$\mathcal{C} = \{(x_j - t)_+, (t - x_j)_+ : t \in \{x_{1j}, \dots, x_{Nj}\}, j = 1, \dots, D\} \quad (16.22)$$

These are 1d linear splines where the knots are at all the observed values for that variable. We consider splines sloping up in both directions; this is called a **reflecting pair**. See Figure 16.7(a).

Let $\mathcal{M}$ represent the current set of basis functions. We initialize by using $\mathcal{M} = \{1\}$. We consider creating a new basis function pair by multiplying an $h_m \in \mathcal{M}$ with one of the reflecting pairs in $\mathcal{C}$. For example, we might initially get

$$f(\mathbf{x}) = 25 - 4(x_1 - 5)_+ + 20(5 - x_1)_+ \quad (16.23)$$

obtained by multiplying $h_0(\mathbf{x}) = 1$ with a reflecting pair involving x_1 with knot $t = 5$. This pair is added to $\mathcal{M}$. See Figure 16.7(b). At the next step, we might create a model such as

$$\begin{aligned} f(\mathbf{x}) = & 2 - 2(x_1 - 5)_+ + 3(5 - x_1)_+ \\ & - (x_2 - 10)_+ \times (5 - x_1) + -1.2(10 - x_2)_+ \times (5 - x_1)_+ \end{aligned} \quad (16.24)$$

obtained by multiplying $(5 - x_1)_+$ from $\mathcal{M}$ by the new reflecting pair $(x_2 - 10)_+$ and $(10 - x_2)_+$. This new function is shown in Figure 16.7(c).

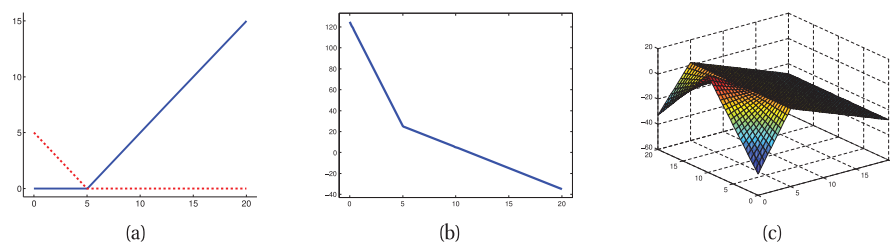


Figure 16.7 (a) Linear spline function with a knot at 5. Solid blue: $(x - 5)_+$. Dotted red: $(5 - x)_+$. (b) A MARS model in 1d given by Equation 16.23. (c) A simple MARS model in 2d given by Equation 16.24. Figure generated by `marsDemo`.

We proceed in this way until the model becomes very large. (We may impose an upper bound on the order of interactions.) Then we prune backwards, at each step eliminating the basis function that causes the smallest increase in the residual error, until the CV error stops improving.

The whole procedure is closely related to CART. To see this, suppose we replace the piecewise linear basis functions by step functions $\mathbb{I}(x_j > t)$ and $\mathbb{I}(x_j < t)$. Multiplying by a pair of reflected step functions is equivalent to splitting a node. Now suppose we impose the constraint that once a variable is involved in a multiplication by a candidate term, that variable gets replaced by the interaction, so the original variable is no longer available. This ensures that a variable can not be split more than once, thus guaranteeing that the resulting model can be represented as a tree. In this case, the MARS growing strategy is the same as the CART growing strategy.

16.4 Boosting

Boosting (Schapire and Freund 2012) is a greedy algorithm for fitting adaptive basis-function models of the form in Equation 16.3, where the ϕ_m are generated by an algorithm called a **weak learner** or a **base learner**. The algorithm works by applying the weak learner sequentially to weighted versions of the data, where more weight is given to examples that were misclassified by earlier rounds.

This weak learner can be any classification or regression algorithm, but it is common to use a CART model. In 1998, the late Leo Breiman called boosting, where the weak learner is a shallow decision tree, the “best off-the-shelf classifier in the world” (Hastie et al. 2009, p340). This is supported by an extensive empirical comparison of 10 different classifiers in (Caruana and Niculescu-Mizil 2006), who showed that boosted decision trees were the best both in terms of misclassification error and in terms of producing well-calibrated probabilities, as judged by ROC curves. (The second best method was random forests, invented by Breiman; see Section 16.2.5.) By contrast, single decision trees performed very poorly.

Boosting was originally derived in the computational learning theory literature (Schapire 1990; Freund and Schapire 1996), where the focus is binary classification. In these papers, it was proved that one could boost the performance (on the training set) of any weak learner arbitrarily

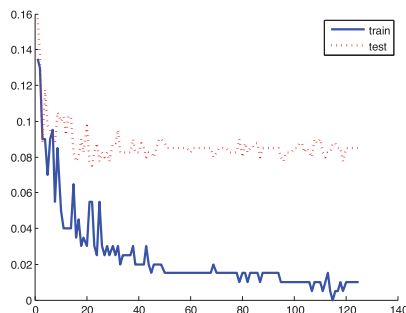


Figure 16.8 Performance of adaboost using a decision stump as a weak learner on the data in Figure 16.10. Training (solid blue) and test (dotted red) error vs number of iterations. Figure generated by `boostingDemo`, written by Richard Stapenhurst.

high, provided the weak learner could always perform slightly better than chance. For example, in Figure 16.8, we plot the training and test error for boosted decision stumps on a 2d dataset shown in Figure 16.10. We see that the training set error rapidly goes to near zero. What is more surprising is that the test set error continues to decline even after the training set error has reached zero (although the test set error will eventually go up). Thus boosting is very resistant to overfitting. (Boosted decision stumps form the basis of a very successful face detector (Viola and Jones 2001), which was used to generate the results in Figure 1.6, and which is used in many digital cameras.)

In view of its stunning empirical success, statisticians started to become interested in this method. Breiman (Breiman 1998) showed that boosting can be interpreted as a form of *gradient descent in function space*. This view was then extended in (Friedman et al. 2000), who showed how boosting could be extended to handle a variety of loss functions, including for regression, robust regression, Poisson regression, etc. In this section, we shall present this statistical interpretation of boosting, drawing on the reviews in (Buhlmann and Hothorn 2007) and (Hastie et al. 2009, ch10), which should be consulted for further details.

16.4.1 Forward stagewise additive modeling

The goal of boosting is to solve the following optimization problem:

$$\min_f \sum_{i=1}^N L(y_i, f(\mathbf{x}_i)) \quad (16.25)$$

and $L(y, \hat{y})$ is some loss function, and f is assumed to be an ABM model as in Equation 16.3. Common choices for the loss function are listed in Table 16.1.

If we use squared error loss, the optimal estimate is given by

$$f^*(\mathbf{x}) = \operatorname{argmin}_{f(\mathbf{x})} \mathbb{E}_{y|\mathbf{x}} [(Y - f(\mathbf{x}))^2] = \mathbb{E}[Y|\mathbf{x}] \quad (16.26)$$

Name	Loss	Derivative	f^*	Algorithm
Squared error	$\frac{1}{2}(y_i - f(\mathbf{x}_i))^2$	$y_i - f(\mathbf{x}_i)$	$\mathbb{E}[y \mathbf{x}_i]$	L2Boosting
Absolute error	$ y_i - f(\mathbf{x}_i) $	$\text{sgn}(y_i - f(\mathbf{x}_i))$	$\text{median}(y \mathbf{x}_i)$	Gradient boosting
Exponential loss	$\exp(-\tilde{y}_i f(\mathbf{x}_i))$	$-\tilde{y}_i \exp(-\tilde{y}_i f(\mathbf{x}_i))$	$\frac{1}{2} \log \frac{\pi_i}{1-\pi_i}$	AdaBoost
Logloss	$\log(1 + e^{-\tilde{y}_i f(\mathbf{x}_i)})$	$y_i - \pi_i$	$\frac{1}{2} \log \frac{\pi_i}{1-\pi_i}$	LogitBoost

Table 16.1 Some commonly used loss functions, their gradients, their population minimizers f^* , and some algorithms to minimize the loss. For binary classification problems, we assume $\tilde{y}_i \in \{-1, +1\}$, $y_i \in \{0, 1\}$ and $\pi_i = \text{sigm}(2f(\mathbf{x}_i))$. For regression problems, we assume $y_i \in \mathbb{R}$. Adapted from (Hastie et al. 2009, p360) and (Buhlmann and Hothorn 2007, p483).

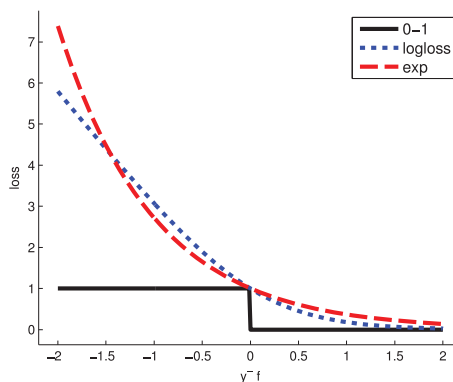


Figure 16.9 Illustration of various loss functions for binary classification. The horizontal axis is the margin $y\eta$, the vertical axis is the loss. The log loss uses log base 2. Figure generated by `hingeLossPlot`.

as we showed in Section 5.7.1.3. Of course, this cannot be computed in practice since it requires knowing the true conditional distribution $p(y|\mathbf{x})$. Hence this is sometimes called the **population minimizer**, where the expectation is interpreted in a frequentist sense. Below we will see that boosting will try to approximate this conditional expectation.

For binary classification, the obvious loss is 0-1 loss, but this is not differentiable. Instead it is common to use logloss, which is a convex upper bound on 0-1 loss, as we showed in Section 6.5.5. In this case, one can show that the optimal estimate is given by

$$f^*(\mathbf{x}) = \frac{1}{2} \log \frac{p(\tilde{y} = 1|\mathbf{x})}{p(\tilde{y} = -1|\mathbf{x})} \quad (16.27)$$

where $\tilde{y} \in \{-1, +1\}$. One can generalize this framework to the multiclass case, but we will not discuss that here.

An alternative convex upper bound is **exponential loss**, defined by

$$L(\tilde{y}, f) = \exp(-\tilde{y}f) \quad (16.28)$$

See Figure 16.9 for a plot. This will have some computational advantages over the logloss, to be discussed below. It turns out that the optimal estimate for this loss is also $f^*(\mathbf{x}) =$

$\frac{1}{2} \log \frac{p(\tilde{y}=1|\mathbf{x})}{p(\tilde{y}=-1|\mathbf{x})}$. To see this, we can just set the derivative of the expected loss (for each $\mathbf{x}$) to zero:

$$\frac{\partial}{\partial f(\mathbf{x})} \mathbb{E} [e^{-\tilde{y}f(\mathbf{x})} | \mathbf{x}] = \frac{\partial}{\partial f(\mathbf{x})} [p(\tilde{y}=1|\mathbf{x})e^{-f(\mathbf{x})} + p(\tilde{y}=-1|\mathbf{x})e^{f(\mathbf{x})}] \quad (16.29)$$

$$= -p(\tilde{y}=1|\mathbf{x})e^{-f(\mathbf{x})} + p(\tilde{y}=-1|\mathbf{x})e^{f(\mathbf{x})} \quad (16.30)$$

$$= 0 \Rightarrow \frac{p(\tilde{y}=1|\mathbf{x})}{p(\tilde{y}=-1|\mathbf{x})} = e^{2f(\mathbf{x})} \quad (16.31)$$

So in both cases, we can see that boosting should try to approximate (half) the log-odds ratio.

Since finding the optimal f is hard, we shall tackle it sequentially. We initialise by defining

$$f_0(\mathbf{x}) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, f(\mathbf{x}_i; \gamma)) \quad (16.32)$$

For example, if we use squared error, we can set $f_0(\mathbf{x}) = \bar{y}$, and if we use log-loss or exponential loss, we can set $f_0(\mathbf{x}) = \frac{1}{2} \log \frac{\hat{\pi}}{1-\hat{\pi}}$, where $\hat{\pi} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}(y_i = 1)$. We could also use a more powerful model for our baseline, such as a GLM.

Then at iteration m , we compute

$$(\beta_m, \gamma_m) = \operatorname{argmin}_{\beta, \gamma} \sum_{i=1}^N L(y_i, f_{m-1}(\mathbf{x}_i) + \beta \phi(\mathbf{x}_i; \gamma)) \quad (16.33)$$

and then we set

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \beta_m \phi(\mathbf{x}; \gamma_m) \quad (16.34)$$

The key point is that we do not go back and adjust earlier parameters. This is why the method is called **forward stagewise additive modeling**.

We continue this for a fixed number of iterations M . In fact M is the main tuning parameter of the method. Often we pick it by monitoring the performance on a separate validation set, and then stopping once performance starts to decrease; this is called **early stopping**. Alternatively, we can use model selection criteria such as AIC or BIC (see e.g., (Buhlmann and Hothorn 2007) for details).

In practice, better (test set) performance can be obtained by performing “partial updates” of the form

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu \beta_m \phi(\mathbf{x}; \gamma_m) \quad (16.35)$$

Here $0 < \nu \leq 1$ is a step-size parameter. In practice it is common to use a small value such as $\nu = 0.1$. This is called **shrinkage**.

Below we discuss how to solve the subproblem in Equation 16.33. This will depend on the form of loss function. However, it is independent of the form of weak learner.

16.4.2 L2boosting

Suppose we used squared error loss. Then at step m the loss has the form

$$L(y_i, f_{m-1}(\mathbf{x}_i) + \beta \phi(\mathbf{x}_i; \gamma)) = (r_{im} - \phi(\mathbf{x}_i; \gamma))^2 \quad (16.36)$$

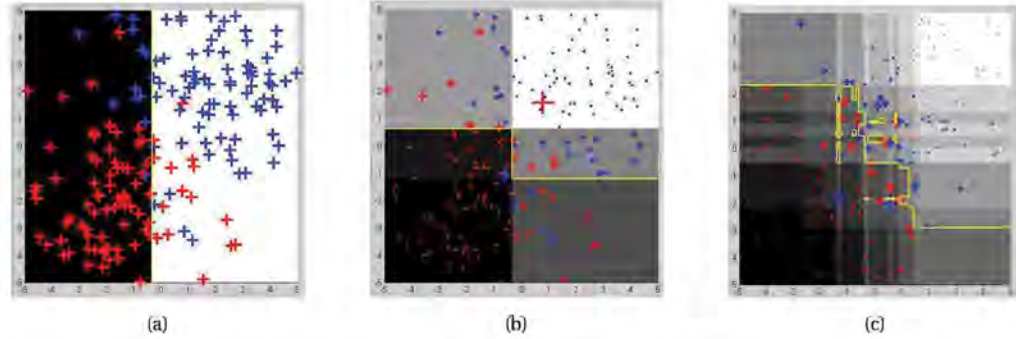


Figure 16.10 Example of adaboost using a decision stump as a weak learner. The degree of blackness represents the confidence in the red class. The degree of whiteness represents the confidence in the blue class. The size of the datapoints represents their weight. Decision boundary is in yellow. (a) After 1 round. (b) After 3 rounds. (c) After 120 rounds. Figure generated by `boostingDemo`, written by Richard Stapenhurst.

where $r_{im} \triangleq y_i - f_{m-1}(\mathbf{x}_i)$ is the current residual, and we have set $\beta = 1$ without loss of generality. Hence we can find the new basis function by using the weak learner to predict $\mathbf{r}_m$. This is called **L2boosting**, or **least squares boosting** (Buhlmann and Yu 2003). In Section 16.4.6, we will see that this method, with a suitable choice of weak learner, can be made to give the same results as LARS, which can be used to perform variable selection (see Section 13.4.2).

16.4.3 AdaBoost

Consider a binary classification problem with exponential loss. At step m we have to minimize

$$L_m(\phi) = \sum_{i=1}^N \exp[-\tilde{y}_i(f_{m-1}(\mathbf{x}_i) + \beta\phi(\mathbf{x}_i))] = \sum_{i=1}^N w_{i,m} \exp(-\beta\tilde{y}_i\phi(\mathbf{x}_i)) \quad (16.37)$$

where $w_{i,m} \triangleq \exp(-\tilde{y}_i f_{m-1}(\mathbf{x}_i))$ is a weight applied to datacase i , and $\tilde{y}_i \in \{-1, +1\}$. We can rewrite this objective as follows:

$$L_m = e^{-\beta} \sum_{\tilde{y}_i = \phi(\mathbf{x}_i)} w_{i,m} + e^{\beta} \sum_{\tilde{y}_i \neq \phi(\mathbf{x}_i)} w_{i,m} \quad (16.38)$$

$$= (e^{\beta} - e^{-\beta}) \sum_{i=1}^N w_{i,m} \mathbb{I}(\tilde{y}_i \neq \phi(\mathbf{x}_i)) + e^{-\beta} \sum_{i=1}^N w_{i,m} \quad (16.39)$$

Consequently the optimal function to add is

$$\phi_m = \underset{\phi}{\operatorname{argmin}} w_{i,m} \mathbb{I}(\tilde{y}_i \neq \phi(\mathbf{x}_i)) \quad (16.40)$$

This can be found by applying the weak learner to a weighted version of the dataset, with weights $w_{i,m}$. Substituting ϕ_m into L_m and solving for β we find

$$\beta_m = \frac{1}{2} \log \frac{1 - \operatorname{err}_m}{\operatorname{err}_m} \quad (16.41)$$

where

$$\text{err}_m = \frac{\sum_{i=1}^N w_i \mathbb{I}(\tilde{y}_i \neq \phi_m(\mathbf{x}_i))}{\sum_{i=1}^N w_{i,m}} \quad (16.42)$$

The overall update becomes

$$f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \beta_m \phi_m(\mathbf{x}) \quad (16.43)$$

With this, the weights at the next iteration become

$$w_{i,m+1} = w_{i,m} e^{-\beta_m \tilde{y}_i \phi_m(\mathbf{x}_i)} \quad (16.44)$$

$$= w_{i,m} e^{\beta_m (2\mathbb{I}(\tilde{y}_i \neq \phi_m(\mathbf{x}_i)) - 1)} \quad (16.45)$$

$$= w_{i,m} e^{2\beta_m \mathbb{I}(\tilde{y}_i \neq \phi_m(\mathbf{x}_i))} e^{-\beta_m} \quad (16.46)$$

where we exploited the fact that $-\tilde{y}_i \phi_m(\mathbf{x}_i) = -1$ if $\tilde{y}_i = \phi_m(\mathbf{x}_i)$ and $-\tilde{y}_i \phi_m(\mathbf{x}_i) = +1$ otherwise. Since $e^{-\beta_m}$ will cancel out in the normalization step, we can drop it. The result is the algorithm shown in Algorithm 7, known **Adaboost.M1**.⁴

An example of this algorithm in action, using decision stumps as the weak learner, is given in Figure 16.10. We see that after many iterations, we can “carve out” a complex decision boundary. What is rather surprising is that AdaBoost is very slow to overfit, as is apparent in Figure 16.8. See Section 16.4.8 for a discussion of this point.

Algorithm 16.2: Adaboost.M1, for binary classification with exponential loss

```

1  $w_i = 1/N$ ;
2 for  $m = 1 : M$  do
3   Fit a classifier  $\phi_m(\mathbf{x})$  to the training set using weights  $\mathbf{w}$ ;
4   Compute  $\text{err}_m = \frac{\sum_{i=1}^N w_{i,m} \mathbb{I}(\tilde{y}_i \neq \phi_m(\mathbf{x}_i))}{\sum_{i=1}^N w_{i,m}}$ ;
5   Compute  $\alpha_m = \log[(1 - \text{err}_m)/\text{err}_m]$ ;
6   Set  $w_i \leftarrow w_i \exp[\alpha_m \mathbb{I}(\tilde{y}_i \neq \phi_m(\mathbf{x}_i))]$ ;
7 Return  $f(\mathbf{x}) = \text{sgn} \left[ \sum_{m=1}^M \alpha_m \phi_m(\mathbf{x}) \right]$ ;
```

16.4.4 LogitBoost

The trouble with exponential loss is that it puts a lot of weight on misclassified examples, as is apparent from the exponential blowup on the left hand side of Figure 16.9. This makes the method very sensitive to outliers (mis-labeled examples). In addition, $e^{-\tilde{y}f}$ is not the logarithm of any pmf for binary variables $\tilde{y} \in \{-1, +1\}$; consequently we cannot recover probability estimates from $f(\mathbf{x})$.

4. In (Friedman et al. 2000), this is called **discrete AdaBoost**, since it assumes that the base classifier ϕ_m returns a binary class label. If ϕ_m returns a probability instead, a modified algorithm, known as **real AdaBoost**, can be used. See (Friedman et al. 2000) for details.

A natural alternative is to use logloss instead. This only punishes mistakes linearly, as is clear from Figure 16.9. Furthermore, it means that we will be able to extract probabilities from the final learned function, using

$$p(y = 1|\mathbf{x}) = \frac{e^{f(\mathbf{x})}}{e^{-f(\mathbf{x})} + e^{f(\mathbf{x})}} = \frac{1}{1 + e^{-2f(\mathbf{x})}} \quad (16.47)$$

The goal is to minimize the expected log-loss, given by

$$L_m(\phi) = \sum_{i=1}^N \log [1 + \exp (-2\tilde{y}_i(f_{m-1}(\mathbf{x}) + \phi(\mathbf{x}_i)))] \quad (16.48)$$

By performing a Newton upate on this objective (similar to IRLS), one can derive the algorithm shown in Algorithm 8. This is known as **logitBoost** (Friedman et al. 2000). It can be generalized to the multi-class setting, as explained in (Friedman et al. 2000).

Algorithm 16.3: LogitBoost, for binary classification with log-loss

```

1  $w_i = 1/N$ ,  $\pi_i = 1/2$ ;
2 for  $m = 1 : M$  do
3   Compute the working response  $z_i = \frac{y_i^* - \pi_i}{\pi_i(1 - \pi_i)}$ ;
4   Compute the weights  $w_i = \pi_i(1 - \pi_i)$ ;
5    $\phi_m = \operatorname{argmin}_{\phi} \sum_{i=1}^N w_i(z_i - \phi(\mathbf{x}_i))^2$ ;
6   Update  $f(\mathbf{x}) \leftarrow f(\mathbf{x}) + \frac{1}{2}\phi_m(\mathbf{x})$ ;
7   Compute  $\pi_i = 1/(1 + \exp(-2f(\mathbf{x}_i)))$ ;
8 Return  $f(\mathbf{x}) = \operatorname{sgn} \left[ \sum_{m=1}^M \phi_m(\mathbf{x}) \right]$ ;
```

16.4.5 Boosting as functional gradient descent

Rather than deriving new versions of boosting for every different loss function, it is possible to derive a generic version, known as **gradient boosting** (Friedman 2001; Mason et al. 2000). To explain this, imagine minimizing

$$\hat{\mathbf{f}} = \operatorname{argmin}_{\mathbf{f}} L(\mathbf{f}) \quad (16.49)$$

where $\mathbf{f} = (f(\mathbf{x}_1), \dots, f(\mathbf{x}_N))$ are the “parameters”. We will solve this stagewise, using gradient descent. At step m , let $\mathbf{g}_m$ be the gradient of $L(\mathbf{f})$ evaluated at $\mathbf{f} = \mathbf{f}_{m-1}$:

$$g_{im} = \left[\frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)} \right]_{f=f_{m-1}} \quad (16.50)$$

Gradients of some common loss functions are given in Table 16.1. We then make the update

$$\mathbf{f}_m = \mathbf{f}_{m-1} - \rho_m \mathbf{g}_m \quad (16.51)$$

where ρ_m is the step length, chosen by

$$\rho_m = \underset{\rho}{\operatorname{argmin}} L(\mathbf{f}_{m-1} - \rho \mathbf{g}_m) \quad (16.52)$$

This is called **functional gradient descent**.

In its current form, this is not much use, since it only optimizes f at a fixed set of N points, so we do not learn a function that can generalize. However, we can modify the algorithm by fitting a weak learner to approximate the negative gradient signal. That is, we use this update

$$\gamma_m = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^N (-g_{im} - \phi(\mathbf{x}_i; \gamma))^2 \quad (16.53)$$

The overall algorithm is summarized in Algorithm 6. (We have omitted the line search step, which is not strictly necessary, as argued in (Buhlmann and Hothorn 2007).)

Algorithm 16.4: Gradient boosting

```

1 Initialize  $f_0(\mathbf{x}) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^N L(y_i, \phi(\mathbf{x}_i; \gamma))$ ;
2 for  $m = 1 : M$  do
3   Compute the gradient residual using  $r_{im} = - \left[ \frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)} \right]_{f(\mathbf{x}_i) = f_{m-1}(\mathbf{x}_i)}$ ;
4   Use the weak learner to compute  $\gamma_m$  which minimizes  $\sum_{i=1}^N (r_{im} - \phi(\mathbf{x}_i; \gamma_m))^2$ ;
5   Update  $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu \phi(\mathbf{x}; \gamma_m)$ ;
6 Return  $f(\mathbf{x}) = f_M(\mathbf{x})$ 
```

If we apply this algorithm using squared loss, we recover L2Boosting. If we apply this algorithm to log-loss, we get an algorithm known as **BinomialBoost** (Buhlmann and Hothorn 2007). The advantage of this over LogitBoost is that it does not need to be able to do weighted fitting: it just applies any black-box regression model to the gradient vector. Also, it is relatively easy to extend to the multi-class case (see (Hastie et al. 2009, p387)). We can also apply this algorithm to other loss functions, such as the Huber loss (Section 7.4), which is more robust to outliers than squared error loss.

16.4.6 Sparse boosting

Suppose we use as our weak learner the following algorithm: search over all possible variables $j = 1 : D$, and pick the one $j(m)$ that best predicts the residual vector:

$$j(m) = \underset{j}{\operatorname{argmin}} \sum_{i=1}^N (r_{im} - \hat{\beta}_{jm} x_{ij})^2 \quad (16.54)$$

$$\hat{\beta}_{jm} = \frac{\sum_{i=1}^N x_{ij} r_{im}}{\sum_{i=1}^N x_{ij}^2} \quad (16.55)$$

$$\phi_m(\mathbf{x}) = \hat{\beta}_{j(m),m} x_{j(m)} \quad (16.56)$$

This method, which is known as **sparse boosting** (Buhlmann and Yu 2006), is identical to the matching pursuit algorithm discussed in Section 13.2.3.1.

It is clear that this will result in a sparse estimate, at least if M is small. To see this, let us rewrite the update as follows:

$$\beta_m := \beta_{m-1} + \nu(0, \dots, 0, \hat{\beta}_{j(m),m}, 0, \dots, 0) \quad (16.57)$$

where the non-zero entry occurs in location $j(m)$. This is known as **forward stagewise linear regression** (Hastie et al. 2009, p608), which becomes equivalent to the LAR algorithm discussed in Section 13.4.2 as $\nu \rightarrow 0$. Increasing the number of steps m in boosting is analogous to decreasing the regularization penalty λ . If we modify boosting to allow some variable deletion steps (Zhao and Yu 2007), we can make it equivalent to the LARS algorithm, which computes the full regularization path for the lasso problem. The same algorithm can be used for sparse logistic regression, by simply modifying the residual to be the appropriate negative gradient.

Now consider a weak learner that is similar to the above, except it uses a smoothing spline instead of linear regression when mapping from x_j to the residual. The result is a sparse generalized additive model (see Section 16.3). It can obviously be extended to pick pairs of variables at a time. The resulting method often works much better than MARS (Buhlmann and Yu 2006).

16.4.7 Multivariate adaptive regression trees (MART)

It is quite common to use CART models as weak learners. It is usually advisable to use a shallow tree, so that the variance is low. Even though the bias will be high (since a shallow tree is likely to be far from the “truth”), this will be compensated for in subsequent rounds of boosting.

The height of the tree is an additional tuning parameter (in addition to M , the number of rounds of boosting, and ν , the shrinkage factor). Suppose we restrict to trees with J leaves. If $J = 2$, we get a stump, which can only split on a single variable. If $J = 3$, we allow for two-variable interactions, etc. In general, it is recommended (e.g., in (Hastie et al. 2009, p363) and (Caruana and Niculescu-Mizil 2006)) to use $J \approx 6$.

If we combine the gradient boosting algorithm with (shallow) regression trees, we get a model known as **MART**, which stands for “multivariate adaptive regression trees”. This actually includes a slight refinement to the basic gradient boosting algorithm: after fitting a regression tree to the residual (negative gradient), we re-estimate the parameters at the leaves of the tree to minimize the loss:

$$\gamma_{jm} = \underset{\gamma}{\operatorname{argmin}} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(\mathbf{x}_i) + \gamma) \quad (16.58)$$

where R_{jm} is the region for leaf j in the m 'th tree, and γ_{jm} is the corresponding parameter (the mean response of y for regression problems, or the most probable class label for classification problems).

16.4.8 Why does boosting work so well?

We have seen that boosting works very well, especially for classifiers. There are two main reasons for this. First, it can be seen as a form of ℓ_1 regularization, which is known to help

prevent overfitting by eliminating “irrelevant” features. To see this, imagine pre-computing all possible weak-learners, and defining a feature vector of the form $\phi(\mathbf{x}) = [\phi_1(\mathbf{x}), \dots, \phi_K(\mathbf{x})]$. We could use ℓ_1 regularization to select a subset of these. Alternatively we can use boosting, where at each step, the weak learner creates a new ϕ_k on the fly. It is possible to combine boosting and ℓ_1 regularization, to get an algorithm known as **L1-Adaboost** (Duchi and Singer 2009). Essentially this method greedily adds the best features (weak learners) using boosting, and then prunes off irrelevant ones using ℓ_1 regularization.

Another explanation has to do with the concept of margin, which we introduced in Section 14.5.2.2. (Schapire et al. 1998; Ratsch et al. 2001) proved that AdaBoost maximizes the margin on the training data. (Rosset et al. 2004) generalized this to other loss functions, such as log-loss.

16.4.9 A Bayesian view

So far, our presentation of boosting has been very frequentist, since it has focussed on greedily minimizing loss functions. A likelihood interpretation of the algorithm was given in (Neal and MacKay 1998; Meek et al. 2002). The idea is to consider a mixture of experts model of the form

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \sum_{m=1}^M \pi_m p(y|\mathbf{x}, \gamma_m) \quad (16.59)$$

where each expert $p(y|\mathbf{x}, \gamma_m)$ is like a weak learner. We usually fit all M experts at once using EM, but we can imagine a sequential scheme, whereby we only update the parameters for one expert at a time. In the E step, the posterior responsibilities will reflect how well the existing experts explain a given data point; if this is a poor fit, these data points will have more influence on the next expert that is fitted. (This view naturally suggest a way to use a boosting-like algorithm for unsupervised learning: we simply sequentially fit mixture models, instead of mixtures of experts.)

Notice that this is a rather “broken” MLE procedure, since it never goes back to update the parameters of an old expert. Similarly, if boosting ever wants to change the weight assigned to a weak learner, the only way to do this is to add the weak learner again with a new weight. This can result in unnecessarily large models. By contrast, the BART model (Chipman et al. 2006, 2010) uses a Bayesian version of backfitting to fit a small sum of weak learners (typically trees).

16.5 Feedforward neural networks (multilayer perceptrons)

A **feedforward neural network**, aka **multi-layer perceptron (MLP)**, is a series of logistic regression models stacked on top of each other, with the final layer being either another logistic regression or a linear regression model, depending on whether we are solving a classification or regression problem. For example, if we have two layers, and we are solving a regression problem, the model has the form

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{w}^T \mathbf{z}(\mathbf{x}), \sigma^2) \quad (16.60)$$

$$\mathbf{z}(\mathbf{x}) = g(\mathbf{V}\mathbf{x}) = [g(\mathbf{v}_1^T \mathbf{x}), \dots, g(\mathbf{v}_H^T \mathbf{x})] \quad (16.61)$$

where g is a non-linear **activation** or **transfer function** (commonly the logistic function), $\mathbf{z}(\mathbf{x}) = \phi(\mathbf{x}, \mathbf{V})$ is called the **hidden layer** (a deterministic function of the input), H is the

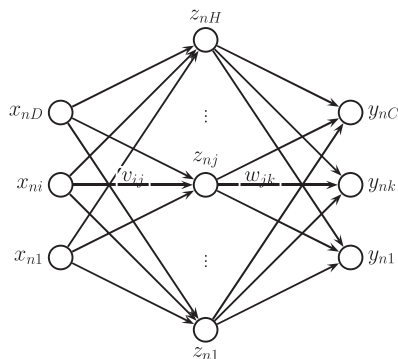


Figure 16.11 A neural network with one hidden layer.

number of **hidden units**, $\mathbf{V}$ is the weight matrix from the inputs to the hidden nodes, and $\mathbf{w}$ is the weight vector from the hidden nodes to the output. It is important that g be non-linear, otherwise the whole model collapses into a large linear regression model of the form $y = \mathbf{w}^T (\mathbf{V}\mathbf{x})$. One can show that an MLP is a **universal approximator**, meaning it can model any suitably smooth function, given enough hidden units, to any desired level of accuracy (Hornik 1991).

To handle binary classification, we pass the output through a sigmoid, as in a GLM:

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \text{Ber}(y|\text{sigm}(\mathbf{w}^T \mathbf{z}(\mathbf{x}))) \quad (16.62)$$

We can easily extend the MLP to predict multiple outputs. For example, in the regression case, we have

$$p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{y}|\mathbf{W} \boldsymbol{\phi}(\mathbf{x}, \mathbf{V}), \sigma^2 \mathbf{I}) \quad (16.63)$$

See Figure 16.11 for an illustration. If we add **mutual inhibition** arcs between the output units, ensuring that only one of them turns on, we can enforce a sum-to-one constraint, which can be used for multi-class classification. The resulting model has the form

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \text{Cat}(y|\mathcal{S}(\mathbf{W}\mathbf{z}(\mathbf{x}))) \quad (16.64)$$

16.5.1 Convolutional neural networks

The purpose of the hidden units is to learn non-linear combinations of the original inputs; this is called **feature extraction** or **feature construction**. These hidden features are then passed as input to the final GLM. This approach is particularly useful for problems where the original input features are not very individually informative. For example, each pixel in an image is not very informative; it is the combination of pixels that tells us what objects are present. Conversely, for a task such as document classification using a bag of words representation, each feature (word count) *is* informative on its own, so extracting “higher order” features is less important. Not suprisingly, then, much of the work in neural networks has been motivated by visual pattern

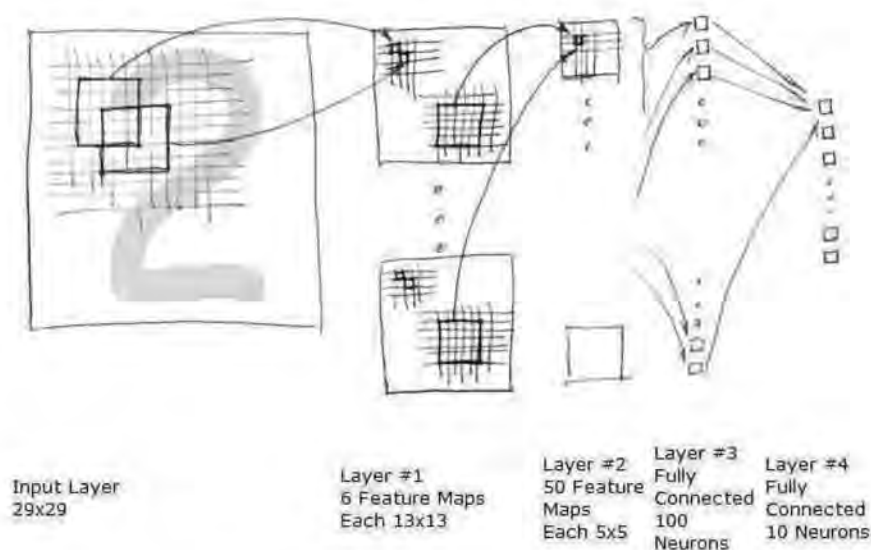


Figure 16.12 The convolutional neural network from (Simard et al. 2003). Source: <http://www.codeproject.com/KB/library/NeuralNetRecognition.aspx>. Used with kind permission of Mike O'Neill.

recognition (e.g., (LeCun et al. 1989)), although they have also been applied to other types of data, including text (e.g., (Collobert and Weston 2008)).

A form of MLP which is particularly well suited to 1d signals like speech or text, or 2d signals like images, is the **convolutional neural network**. This is an MLP in which the hidden units have local **receptive fields** (as in the primary visual cortex), and in which the weights are **tied** or shared across the image, in order to reduce the number of parameters. Intuitively, the effect of such spatial parameter tying is that any useful features that are “discovered” in some portion of the image can be re-used everywhere else without having to be independently learned. The resulting network then exhibits **translation invariance**, meaning it can classify patterns no matter where they occur inside the input image.

Figure 16.12 gives an example of a convolutional network, designed by Simard and colleagues (Simard et al. 2003), with 5 layers (4 layers of adjustable parameters) designed to classify 29×29 gray-scale images of handwritten digits from the MNIST dataset (see Section 1.2.1.3). In layer 1, we have 6 **feature maps** each of which has size 13×13 . Each hidden node in one of these feature maps is computed by convolving the image with a 5×5 weight matrix (sometimes called a kernel), adding a bias, and then passing the result through some form of nonlinearity. There are therefore $13 \times 13 \times 6 = 1014$ neurons in Layer 1, and $(5 \times 5 + 1) \times 6 = 156$ weights. (The “+1” is for the bias.) If we did not share these parameters, there would be $1014 \times 26 = 26,364$ weights at the first layer. In layer 2, we have 50 feature maps, each of which is obtained by convolving each feature map in layer 1 with a 5×5 weight matrix, adding them up, adding a bias, and passing through a nonlinearity. There are therefore $5 \times 5 \times 50 = 1250$ neurons in Layer 2, $(5 \times 5 + 1) \times 6 \times 50 = 7800$ adjustable weights (one kernel for each pair of feature

maps in layers 1 and 2), and $1250 \times 26 = 32,500$ connections. Layer 3 is fully connected to layer 2, and has 100 neurons and $100 \times (1250 + 1) = 125,100$ weights. Finally, layer 4 is also fully connected, and has 10 neurons, and $10 \times (100 + 1) = 1010$ weights. Adding the above numbers, there are a total of 3,215 neurons, 134,066 adjustable weights, and 184,974 connections.

This model is usually trained using stochastic gradient descent (see Section 16.5.4 for details). A single pass over the data set is called an epoch. When Mike O'Neill did these experiments in 2006, he found that a single epoch took about 40 minutes (recall that there are 60,000 training examples in MNIST). Since it took about 30 epochs for the error rate to converge, the total training time was about 20 hours.⁵ Using this technique, he obtained a misclassification rate on the 10,000 test cases of about 1.40%.

To further reduce the error rate, a standard trick is to expand the training set by including **distorted** versions of the original data, to encourage the network to be invariant to small changes that don't affect the identity of the digit. These can be created by applying a random flow field to shift pixels around. See Figure 16.13 for some examples. (If we use online training, such as stochastic gradient descent, we can create these distortions on the fly, rather than having to store them.) Using this technique, Mike O'Neill obtained a misclassification rate on the 10,000 test cases of about 0.74%, which is close to the current state of the art.⁶

Yann Le Cun and colleagues (LeCun et al. 1998) obtained similar performance using a slightly more complicated architecture shown in Figure 16.14. This model is known as **LeNet5**, and historically it came before the model in Figure 16.12. There are two main differences. First, LeNet5 has a **subsampling** layer between each convolutional layer, which either averages or computes the max over each small window in the previous layer, in order to reduce the size, and to obtain a small amount of shift invariance. The convolution and sub-sampling combination was inspired by Hubel and Wiesel's model of simple and complex cells in the visual cortex (Hubel and Wiesel 1962), and it continues to be popular in neurally-inspired models of visual object recognition (Riesenhuber and Poggio 1999). A similar idea first appeared in Fukushima's **neocognitron** (Fukushima 1975), though no globally supervised training algorithm was available at that time.

The second difference between LeNet5 and the Simard architecture is that the final layer is actually an RBF network rather than a more standard sigmoidal or softmax layer. This model gets a test error rate of about 0.95% when trained with no distortions, and 0.8% when trained with distortions. Figure 16.15 shows all 82 errors made by the system. Some are genuinely ambiguous, but several are errors that a person would never make. A web-based demo of the LeNet5 can be found at <http://yann.lecun.com/exdb/lenet/index.html>.

Of course, classifying isolated digits is of limited applicability: in the real world, people usually write strings of digits or other letters. This requires both segmentation and classification. Le Cun and colleagues devised a way to combine convolutional neural networks with a model similar to a conditional random field (described in Section 19.6) to solve this problem. The system was eventually deployed by the US postal service. (See (LeCun et al. 1998) for a more detailed account of the system, which remains one of the best performing systems for this task.)

5. Implementation details: Mike used C++ code and a variety of speedup tricks. He was using standard 2006 era hardware (an Intel Pentium 4 hyperthreaded processor running at 2.8GHz). See <http://www.codeproject.com/KB/Library/NeuralNetRecognition.aspx> for details.

6. A list of various methods, along with their misclassification rates on the MNIST test set, is available from <http://yann.lecun.com/exdb/mnist/>. Error rates within 0.1–0.2% of each other are not statistically significantly different.

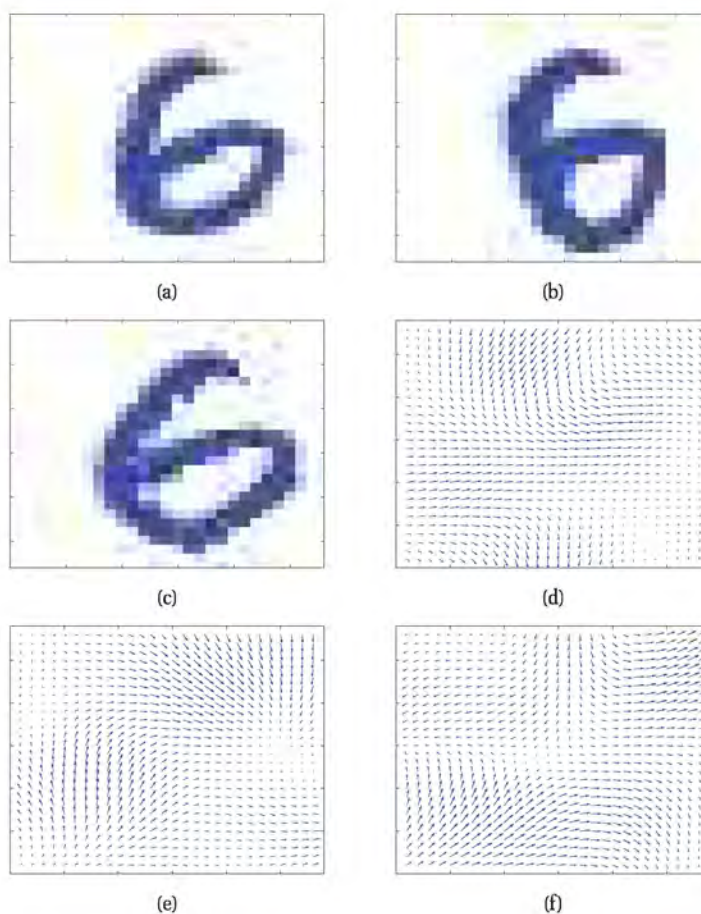


Figure 16.13 Several synthetic warping of a handwritten digit. Based on Figure 5.14 of (Bishop 2006a). Figure generated by `elasticDistortionsDemo`, written by Kevin Swersky.

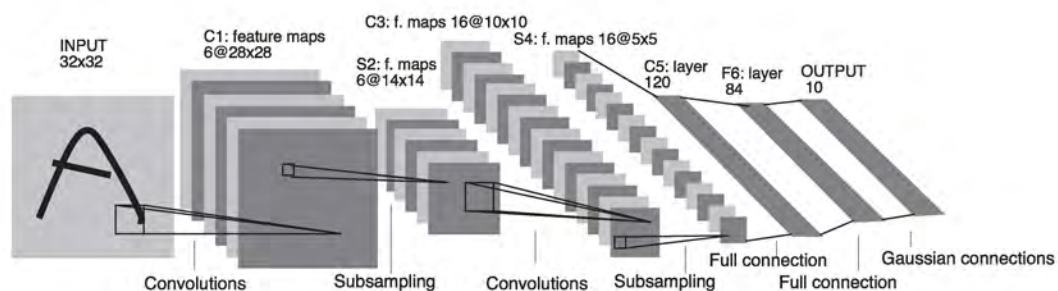


Figure 16.14 LeNet5, a convolutional neural net for classifying handwritten digits. Source: Figure 2 from (LeCun et al. 1998) . Used with kind permission of Yann LeCun.



Figure 16.15 These are the 82 errors made by LeNet5 on the 10,000 test cases of MNIST. Below each image is a label of the form correct-label → estimated-label. Source: Figure 8 of (LeCun et al. 1998). Used with kind permission of Yann LeCun. (Compare to Figure 28.4(b) which shows the results of a deep generative model.)

16.5.2 Other kinds of neural networks

Other network topologies are possible besides the ones discussed above. For example, we can have **skip arcs** that go directly from the input to the output, skipping the hidden layer; we can have sparse connections between the layers; etc. However, the MLP always requires that the weights form a directed acyclic graph. If we allow feedback connections, the model is known as a **recurrent neural network**; this defines a nonlinear dynamical system, but does not have a simple probabilistic interpretation. Such RNN models are currently the best approach for language modeling (i.e., performing word prediction in natural language) (Tomas et al. 2011), significantly outperforming the standard n-gram-based methods discussed in Section 17.2.2.

If we allow symmetric connections between the hidden units, the model is known as a **Hopfield network** or **associative memory**; its probabilistic counterpart is known as a Boltzmann machine (see Section 27.7) and can be used for unsupervised learning.

16.5.3 A brief history of the field

Neural networks have been the subject of great interest for many decades, due to the desire to understand the brain, and to build learning machines. It is not possible to review the entire history here. Instead, we just give a few “edited highlights”.

The field is generally viewed as starting with McCulloch and Pitts (McCulloch and Pitts 1943), who devised a simple mathematical model of the neuron in 1943, in which they approximated the

output as a weighted sum of inputs passed through a threshold function, $y = \mathbb{I}(\sum_i w_i x_i > \theta)$, for some threshold θ . This is similar to a sigmoidal activation function. Frank Rosenblatt invented the perceptron learning algorithm in 1957, which is a way to estimate the parameters of a McCulloch-Pitts neuron (see Section 8.5.4 for details). A very similar model called the **adaline** (for adaptive linear element) was invented in 1960 by Widrow and Hoff.

In 1969, Minsky and Papert (Minsky and Papert 1969) published a famous book called “Perceptrons” in which they showed that such linear models, with no hidden layers, were very limited in their power, since they cannot classify data that is not linearly separable. This considerably reduced interest in the field.

In 1986, Rumelhart, Hinton and Williams (Rumelhart et al. 1986) discovered the backpropagation algorithm (see Section 16.5.4), which allows one to fit models with hidden layers. (The backpropagation algorithm was originally discovered in (Bryson and Ho 1969), and independently in (Werbos 1974); however, it was (Rumelhart et al. 1986) that brought the algorithm to people’s attention.) This spawned a decade of intense interest in these models.

In 1987, Sejnowski and Rosenberg (Sejnowski and Rosenberg 1987) created the famous **NETtalk** system, that learned a mapping from English words to phonetic symbols which could be fed into a speech synthesizer. An audio demo of the system as it learns over time can be found at <http://www.cnl.salk.edu/ParallelNetsPronounce/nettalk.mp3>. The system starts by “babbling” and then gradually learns to pronounce English words. NETtalk learned a **distributed representation** (via its hidden layer) of various sounds, and its success spawned a big debate in psychology between **connectionism**, based on neural networks, and **computationalism**, based on syntactic rules. This debate lives on to some extent in the machine learning community, where there are still arguments about whether learning is best performed using low-level, “neural-like” representations, or using more structured models.

In 1989, Yann Le Cun and others (LeCun et al. 1989) created the famous LeNet system described in Section 16.5.1.

In 1992, the support vector machine (see Section 14.5) was invented (Boser et al. 1992). SVMs provide similar prediction accuracy to neural networks while being considerably easier to train (since they use a convex objective function). This spawned a decade of interest in kernel methods in general.⁷ Note, however, that SVMs do not use adaptive basis functions, so they require a fair amount of human expertise to design the right kernel function.

In 2002, Geoff Hinton invented the contrastive divergence training procedure (Hinton 2002), which provided a way, for the first time, to learn deep networks, by training one layer at a time in an unsupervised fashion (see Section 27.7.2.4 for details). This in turn has spawned renewed interest in neural networks over the last few years (see Chapter 28).

16.5.4 The backpropagation algorithm

Unlike a GLM, the NLL of an MLP is a non-convex function of its parameters. Nevertheless, we can find a locally optimal ML or MAP estimate using standard gradient-based optimization methods. Since MLPs have lots of parameters, they are often trained on very large data sets.

7. It became part of the folklore during the 1990s that to get published in the top machine learning conference known as NIPS, which stands for “neural information processing systems”, it was important to ensure your paper did not contain the word “neural network”!

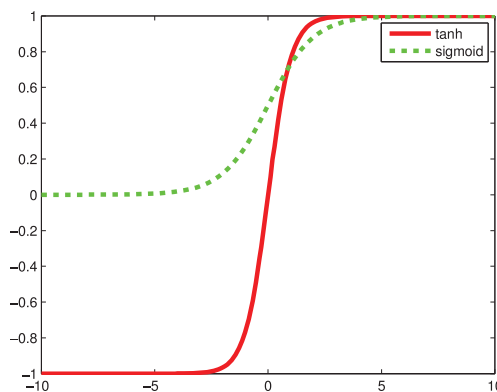


Figure 16.16 Two possible activation functions. $\tanh$ maps $\mathbb{R}$ to $[-1, +1]$ and is the preferred nonlinearity for the hidden nodes. sigm maps $\mathbb{R}$ to $[0, 1]$ and is the preferred nonlinearity for binary nodes at the output layer. Figure generated by `tanhPlot`.

Consequently it is common to use first-order online methods, such as stochastic gradient descent (Section 8.5.2), whereas GLMs are usually fit with IRLS, which is a second-order offline method.

We now discuss how to compute the gradient vector of the NLL by applying the chain rule of calculus. The resulting algorithm is known as **backpropagation**, for reasons that will become apparent.

For notational simplicity, we shall assume a model with just one hidden layer. It is helpful to distinguish the pre- and post-synaptic values of a neuron, that is, before and after we apply the nonlinearity. Let $\mathbf{x}_n$ be the n 'th input, $\mathbf{a}_n = \mathbf{V}\mathbf{x}_n$ be the pre-synaptic hidden layer, and $\mathbf{z}_n = g(\mathbf{a}_n)$ be the post-synaptic hidden layer, where g is some **transfer function**. We typically use $g(a) = \text{sigm}(a)$, but we may also use $g(a) = \tanh(a)$: see Figure 16.16 for a comparison. (When the input to sigm or $\tanh$ is a vector, we assume it is applied component-wise.)

We now convert this hidden layer to the output layer as follows. Let $\mathbf{b}_n = \mathbf{W}\mathbf{z}_n$ be the pre-synaptic output layer, and $\hat{\mathbf{y}}_n = h(\mathbf{b}_n)$ be the post-synaptic output layer, where h is another nonlinearity, corresponding to the canonical link for the GLM. (We reserve the notation $\mathbf{y}_n$, without the hat, for the output corresponding to the n 'th training case.) For a regression model, we use $h(\mathbf{b}) = \mathbf{b}$; for binary classification, we use $h(\mathbf{b}) = [\text{sigm}(b_1), \dots, \text{sigm}(b_c)]$; for multi-class classification, we use $h(\mathbf{b}) = \mathcal{S}(\mathbf{b})$.

We can write the overall model as follows:

$$\mathbf{x}_n \xrightarrow{\mathbf{V}} \mathbf{a}_n \xrightarrow{g} \mathbf{z}_n \xrightarrow{\mathbf{W}} \mathbf{b}_n \xrightarrow{h} \hat{\mathbf{y}}_n \quad (16.65)$$

The parameters of the model are $\boldsymbol{\theta} = (\mathbf{V}, \mathbf{W})$, the first and second layer weight matrices. Offset or bias terms can be accommodated by clamping an element of $\mathbf{x}_n$ and $\mathbf{z}_n$ to 1.⁸

8. In the regression setting, we can easily estimate the variance of the output noise using the empirical variance of the residual errors, $\hat{\sigma}^2 = \frac{1}{N} \|\hat{\mathbf{y}}(\hat{\boldsymbol{\theta}}) - \mathbf{y}\|^2$, after training is complete. There will be one value of σ^2 for each output node, if we are performing multi-target regression, as we usually assume.

In the regression case, with K outputs, the NLL is given by the squared error:

$$J(\boldsymbol{\theta}) = - \sum_n \sum_k (\hat{y}_{nk}(\boldsymbol{\theta}) - y_{nk})^2 \quad (16.66)$$

In the classification case, with K classes, the NLL is given by the cross entropy

$$J(\boldsymbol{\theta}) = - \sum_n \sum_k y_{nk} \log \hat{y}_{nk}(\boldsymbol{\theta}) \quad (16.67)$$

Our task is to compute $\nabla_{\boldsymbol{\theta}} J$. We will derive this for each n separately; the overall gradient is obtained by summing over n , although often we just use a mini-batch (see Section 8.5.2).

Let us start by considering the output layer weights. We have

$$\nabla_{\mathbf{w}_k} J_n = \frac{\partial J_n}{\partial b_{nk}} \nabla_{\mathbf{w}_k} b_{nk} = \frac{\partial J_n}{\partial b_{nk}} \mathbf{z}_n \quad (16.68)$$

since $b_{nk} = \mathbf{w}_k^T \mathbf{z}_n$. Assuming h is the canonical link function for the output GLM, then Equation 9.91 tells us that

$$\frac{\partial J_n}{\partial b_{nk}} \triangleq \delta_{nk}^w = (\hat{y}_{nk} - y_{nk}) \quad (16.69)$$

which is the error signal. So the overall gradient is

$$\nabla_{\mathbf{w}_k} J_n = \delta_{nk}^w \mathbf{z}_n \quad (16.70)$$

which is the pre-synaptic input to the output layer, namely $\mathbf{z}_n$, times the error signal, namely δ_{nk}^w .

For the input layer weights, we have

$$\nabla_{\mathbf{v}_j} J_n = \frac{\partial J_n}{\partial a_{nj}} \nabla_{\mathbf{v}_j} a_{nj} \triangleq \delta_{nj}^v \mathbf{x}_n \quad (16.71)$$

where we exploited the fact that $a_{nj} = \mathbf{v}_j^T \mathbf{x}_n$. All that remains is to compute the first level error signal δ_{nj}^v . We have

$$\delta_{nj}^v = \frac{\partial J_n}{\partial a_{nj}} = \sum_{k=1}^K \frac{\partial J_n}{\partial b_{nk}} \frac{\partial b_{nk}}{\partial a_{nj}} = \sum_{k=1}^K \delta_{nk}^w \frac{\partial b_{nk}}{\partial a_{nj}} \quad (16.72)$$

Now

$$b_{nk} = \sum_j w_{kj} g(a_{nj}) \quad (16.73)$$

so

$$\frac{\partial b_{nk}}{\partial a_{nj}} = w_{kj} g'(a_{nj}) \quad (16.74)$$

where $g'(a) = \frac{d}{da} g(a)$. For tanh units, $g'(a) = \frac{d}{da} \tanh(a) = 1 - \tanh^2(a) = \text{sech}^2(a)$, and for sigmoid units, $g'(a) = \frac{d}{da} \sigma(a) = \sigma(a)(1 - \sigma(a))$. Hence

$$\delta_{nj}^v = \sum_{k=1}^K \delta_{nk}^w w_{kj} g'(a_{nj}) \quad (16.75)$$

Thus the layer 1 errors can be computed by passing the layer 2 errors back through the $\mathbf{W}$ matrix; hence the term “backpropagation”. The key property is that we can compute the gradients locally: each node only needs to know about its immediate neighbors. This is supposed to make the algorithm “neurally plausible”, although this interpretation is somewhat controversial.

Putting it all together, we can compute all the gradients as follows: we first perform a forwards pass to compute $\mathbf{a}_n$, $\mathbf{z}_n$, $\mathbf{b}_n$ and $\hat{\mathbf{y}}_n$. We then compute the error for the output layer, $\delta_n^{(2)} = \hat{\mathbf{y}}_n - \mathbf{y}_n$, which we pass backwards through $\mathbf{W}$ using Equation 16.75 to compute the error for the hidden layer, $\delta_n^{(1)}$. We then compute the overall gradient as follows:

$$\nabla_{\theta} J(\theta) = \sum_n [\delta_n^v \mathbf{x}_n, \delta_n^w \mathbf{z}_n] \quad (16.76)$$

16.5.5 Identifiability

It is easy to see that the parameters of a neural network are not identifiable. For example, we can change the sign of the weights going into one of the hidden units, so long as we change the sign of all the weights going out of it; these effects cancel, since $\tanh$ is an odd function, so $\tanh(-a) = -\tanh(a)$. There will be H such sign flip symmetries, leading to 2^H equivalent settings of the parameters. Similarly, we can change the identity of the hidden units without affecting the likelihood. There are $H!$ such permutations. The total number of equivalent parameter settings (with the same likelihood) is therefore $H!2^H$.

In addition, there may be local minima due to the non-convexity of the NLL. This can be a more serious problem, although with enough data, these local optima are often quite “shallow”, and simple stochastic optimization methods can avoid them. In addition, it is common to perform multiple restarts, and to pick the best solution, or to average over the resulting predictions. (It does not make sense to average the parameters themselves, since they are not identifiable.)

16.5.6 Regularization

As usual, the MLE can overfit, especially if the number of nodes is large. A simple way to prevent this is called **early stopping**, which means stopping the training procedure when the error on the validation set first starts to increase. This method works because we usually initialize from small random weights, so the model is initially simple (since the $\tanh$ and sigm functions are nearly linear near the origin). As training progresses, the weights become larger, and the model becomes nonlinear. Eventually it will overfit.

Another way to prevent overfitting, that is more in keeping with the approaches used elsewhere in this book, is to impose a prior on the parameters, and then use MAP estimation. It is standard to use a $\mathcal{N}(0, \alpha^{-1} \mathbf{I})$ prior (equivalent to ℓ_2 regularization), where α is the precision (strength) of the prior. In the neural networks literature, this is called **weight decay**, since it encourages small weights, and hence simpler models. The penalized NLL objective becomes

$$J(\theta) = - \sum_{n=1}^N \log p(y_n | \mathbf{x}_n, \theta) + \frac{\alpha}{2} \left[\sum_{ij} v_{ij}^2 + \sum_{jk} w_{jk}^2 \right] \quad (16.77)$$