Learning with Trees

We are now going to consider a rather different approach to machine learning, starting with one of the most common and powerful data structures in the whole of computer science: the binary tree. The computational cost of making the tree is fairly low, but the cost of using it is even lower: $\mathcal{O}(\log N)$, where N is the number of datapoints. This is important for machine learning, since querying the trained algorithm should be as fast as possible since it happens more often, and the result is often wanted immediately. This is sufficient to make trees seem attractive for machine learning. However, they do have other benefits, such as the fact that they are easy to understand (following a tree to get a classification answer is transparent, which makes people trust it more than getting an answer from a 'black box' neural network).

For these reasons, classification by decision trees has grown in popularity over recent years. You are very likely to have been subjected to decision trees if you've ever phoned a helpline, for example for computer faults. The phone operators are guided through the decision tree by your answers to their questions.

The idea of a decision tree is that we break classification down into a set of choices about each feature in turn, starting at the root (base) of the tree and progressing down to the leaves, where we receive the classification decision. The trees are very easy to understand, and can even be turned into a set of if-then rules, suitable for use in a rule induction system.

In terms of optimisation and search, decision trees use a greedy heuristic to perform search, evaluating the possible options at the current stage of learning and making the one that seems optimal at that point. This works well a surprisingly large amount of the time.

12.1 USING DECISION TREES

As a student it can be difficult to decide what to do in the evening. There are four things that you actually quite enjoy doing, or have to do: going to the pub, watching TV, going to a party, or even (gasp) studying. The choice is sometimes made for you—if you have an assignment due the next day, then you need to study, if you are feeling lazy then the pub isn't for you, and if there isn't a party then you can't go to it. You are looking for a nice algorithm that will let you decide what to do each evening without having to think about it every night. Figure 12.1 provides just such an algorithm.

Each evening you start at the top (root) of the tree and check whether any of your friends know about a party that night. If there is one, then you need to go, regardless. Only if there is not a party do you worry about whether or not you have an assignment deadline coming up. If there is a crucial deadline, then you have to study, but if there is nothing that is urgent for the next few days, you think about how you feel. A sudden burst of energy

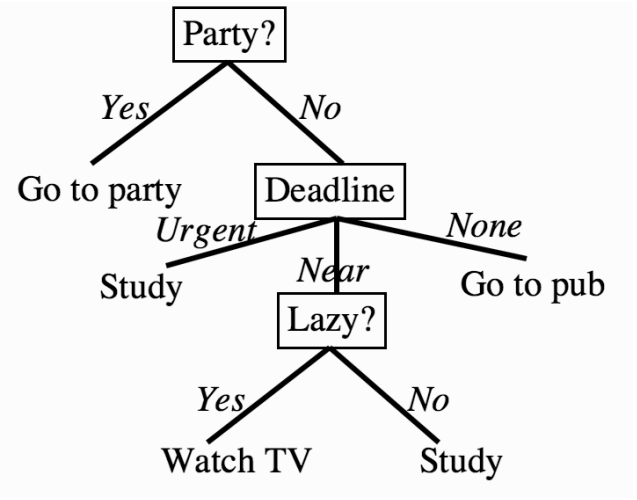

FIGURE 12.1 A simple decision tree to decide how you will spend the evening.

might make you study, but otherwise you'll be slumped in front of the TV indulging your secret love of Shortland Street (or other soap opera of your choice) rather than studying. Of course, near the start of the semester when there are no assignments to do, and you are feeling rich, you'll be in the pub.

One of the reasons that decision trees are popular is that we can turn them into a set of logical disjunctions (`if ... then` rules) that then go into program code very simply—the first part of the tree above can be turned into:

- `if` *there is a party* `then` *go to it*
- `if` *there is not a party* `and` *you have an urgent deadline* `then` *study*
- etc.

That's all that there is to using the decision tree. Compare it to the previous use of this data, with the Naïve Bayes Classifier in Section 2.3.2. The far more interesting part is how to construct the tree from data, and that is the focus of the next section.

12.2 CONSTRUCTING DECISION TREES

In the example above, the three features that we need for the algorithm are the state of your energy level, the date of your nearest deadline, and whether or not there is a party tonight. The question we need to ask is how, based on those features, we can construct the tree. There are a few different decision tree algorithms, but they are almost all variants of the same principle: the algorithms build the tree in a greedy manner starting at the root, choosing the most informative feature at each step. We are going to start by focusing on the most common: Quinlan's ID3, although we'll also mention its extension, known as C4.5, and another known as CART.

There was an important word hidden in the sentence above about how the trees work, which was informative. Choosing which feature to use next in the decision tree can be thought of as playing the game '20 Questions', where you try to elicit the item your opponent is thinking about by asking questions about it. At each stage, you choose a question that gives you the most information given what you know already. Thus, you would ask 'Is it an animal?' before you ask 'Is it a cat?'. The idea is to quantify this question of how much

information is provided to you by knowing certain facts. Encoding this mathematically is the task of information theory.

12.2.1 Quick Aside: Entropy in Information Theory

Information theory was 'born' in 1948 when Claude Shannon published a paper called "A Mathematical Theory of Communication." In that paper, he proposed the measure of information entropy, which describes the amount of impurity in a set of features. The entropy H of a set of probabilities p_i is (for those who know some physics, the relation to physical entropy should be clear):

$$\text{Entropy}(p) = -\sum_i p_i \log_2 p_i, \tag{12.1}$$

where the logarithm is base 2 because we are imagining that we encode everything using binary digits (bits), and we define $0 \log 0 = 0$. A graph of the entropy is given in Figure 12.2. Suppose that we have a set of positive and negative examples of some feature (where the feature can only take 2 values: positive and negative). If all of the examples are positive, then we don't get any extra information from knowing the value of the feature for any particular example, since whatever the value of the feature, the example will be positive. Thus, the entropy of that feature is 0. However, if the feature separates the examples into 50% positive and 50% negative, then the amount of entropy is at a maximum, and knowing about that feature is very useful to us. The basic concept is that it tells us how much *extra* information we would get from knowing the value of that feature. A function for computing the entropy is very simple, as here:

```
def calc_entropy(p):

    if p!=0:
        return -p * np.log2(p)
    else:
        return 0
```

For our decision tree, the best feature to pick as the one to classify on now is the one that gives you the most information, i.e., the one with the highest entropy. After using that feature, we re-evaluate the entropy of each feature and again pick the one with the highest entropy.

Information theory is a very interesting subject. It is possible to download Shannon's 1948 paper from the Internet, and also to find many resources showing where it has been applied. There are now whole journals devoted to information theory because it is relevant to so many areas such as computer and telecommunication networks, machine learning, and data storage. Some further readings in the area are given at the end of the chapter.

12.2.2 ID3

Now that we have a suitable measure for choosing which feature to choose next, entropy, we just have to work out how to apply it. The important idea is to work out how much the entropy of the whole training set would decrease if we choose each particular feature for the next classification step. This is known as the information gain, and it is defined as

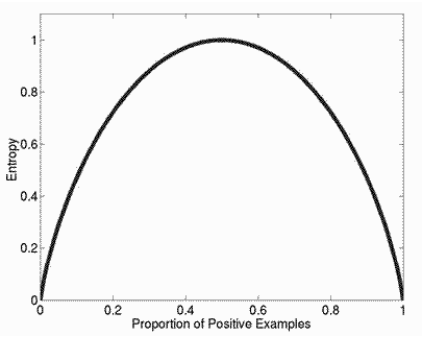

FIGURE 12.2 A graph of entropy, detailing how much information is available from finding out another piece of information given what you already know.

the entropy of the whole set minus the entropy when a particular feature is chosen. This is defined by (where S is the set of examples, F is a possible feature out of the set of all possible ones, and $|S_f|$ is a count of the number of members of S that have value f for feature F):

$$\text{Gain}(S, F) = \text{Entropy}(S) - \sum_{f \in values(F)} \frac{|S_f|}{|S|} \text{Entropy}(S_f). \tag{12.2}$$

As an example, suppose that we have data (with outcomes) $S = \{s_1 = \text{true}, s_2 = \text{false}, s_3 = \text{false}, s_4 = \text{false}\}$ and one feature F that can have values $\{f_1, f_2, f_3\}$. In the example, the feature value for s_1 could be f_2, for s_2 it could be f_2, for s_3, f_3 and for s_4, f_1 then we can calculate the entropy of S as (where $\oplus$ means true, of which we have one example, and $\ominus$ means false, of which we have three examples):

$$\begin{aligned} \text{Entropy}(S) &= -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus \\ &= -\frac{1}{4} \log_2 \frac{1}{4} - \frac{3}{4} \log_2 \frac{3}{4} \\ &= 0.5 + 0.311 = 0.811. \end{aligned} \tag{12.3}$$

The function $\text{Entropy}(S_f)$ is similar, but only computed with the subset of data where feature F has values f.

If you were trying to follow those calculations on a calculator, you might be wondering how to compute $\log_2 p$. The answer is to use the identity $\log_2 p = \ln p / \ln(2)$, where ln is the natural logarithm, which your calculator can produce. NumPy has the `np.log2()` function.

We now want to compute the information gain of F, so we now need to compute each of the values inside the summation in Equation (12.2), $\frac{|S_f|}{|S|}\text{Entropy}(S)$ (in our example, the features are 'Deadline', 'Party', and 'Lazy'):

$$\begin{aligned}
\frac{|S_{f_1}|}{|S|}\text{Entropy}(S_{f_1}) &= \frac{1}{4} \times \left(-\frac{0}{1}\log_2\frac{0}{1} - \frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 0 \qquad (12.4) \\
\frac{|S_{f_2}|}{|S|}\text{Entropy}(S_{f_2}) &= \frac{2}{4} \times \left(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}\right) \\
&= \frac{1}{2} \qquad (12.5) \\
\frac{|S_{f_3}|}{|S|}\text{Entropy}(S_{f_3}) &= \frac{1}{4} \times \left(-\frac{0}{1}\log_2\frac{0}{1} - \frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 0 \qquad (12.6)
\end{aligned}$$

The information gain from adding this feature is the entropy of S minus the sum of the three values above:

$$\text{Gain}(S, F) = 0.811 - (0 + 0.5 + 0) = 0.311. \qquad (12.7)$$

This can be computed in an algorithm using the following function (where lots of the code is to get the relevant data):

```
def calc_info_gain(data,classes,feature):
  gain = 0
  nData = len(data)
  # List the values that feature can take
  values = []
  for datapoint in data:
      if datapoint[feature] not in values:
         values.append(datapoint[feature])

  featureCounts = np.zeros(len(values))
  entropy = np.zeros(len(values))
  valueIndex = 0
  # Find where those values appear in data[feature] and the corresponding ❯
  class
  for value in values:
      dataIndex = 0
      newClasses = []
      for datapoint in data:
         if datapoint[feature]==value:
            featureCounts[valueIndex]+=1
            newClasses.append(classes[dataIndex])
         dataIndex += 1

      # Get the values in newClasses
      classValues = []
      for aclass in newClasses:
         if classValues.count(aclass)==0:
            classValues.append(aclass)
```

```
        classCounts = np.zeros(len(classValues))
        classIndex = 0
        for classValue in classValues:
            for aclass in newClasses:
                if aclass == classValue:
                    classCounts[classIndex]+=1
            classIndex += 1

        for classIndex in range(len(classValues)):
            entropy[valueIndex] += calc_entropy(float(classCounts[classIndex])
            /sum(classCounts))
        gain += float(featureCounts[valueIndex])/nData * entropy[valueIndex]
        valueIndex += 1
    return gain
```

The ID3 algorithm computes this information gain for each feature and chooses the one that produces the highest value. In essence, that is all there is to the algorithm. It searches the space of possible trees in a greedy way by choosing the feature with the highest information gain at each stage. The output of the algorithm is the tree, i.e., a list of nodes, edges, and leaves. As with any tree in computer science, it can be constructed recursively. At each stage the best feature is selected and then removed from the dataset, and the algorithm is recursively called on the rest. The recursion stops when either there is only one class remaining in the data (in which case a leaf is added with that class as its label), or there are no features left, when the most common label in the remaining data is used.

The ID3 Algorithm

- If all examples have the same label:
  - return a leaf with that label
- Else if there are no features left to test:
  - return a leaf with the most common label
- Else:
  - choose the feature $\hat{F}$ that maximises the information gain of S to be the next node using Equation (12.2)
  - add a branch from the node for each possible value f in $\hat{F}$
  - for each branch:
    - calculate S_f by removing $\hat{F}$ from the set of features
    - recursively call the algorithm with S_f, to compute the gain relative to the current set of examples

Owing to the focus on classification for real-world examples, trees are often used with text features rather than numeric values. This makes it rather difficult to use NumPy, and so the sample implementation is pretty well pure Python. It uses a feature of Python that is uncommon in other languages, which is the dictionary in order to hold the tree, which uses the braces {, }, and which is described next before we look at the decision tree implementation.

12.2.3 Implementing Trees and Graphs in Python

Trees are really just a restricted version of graphs, since they both consist of nodes and edges between the nodes. Graphs are a very useful data structure in many different areas of computer science. There are two reasonable ways to represent a graph computationally. One is as an $N \times N$ matrix, where N is the number of nodes in the network. Each element of the matrix is a 1 if there is a link between the two nodes, and a 0 otherwise. The benefit of this approach is that it is easy to give weights to the links by changing the 1s to the values of the weights. The alternative is to store a list of nodes, following each by a list of nodes that it is linked to. Both are fairly natural in Python, with the second making use of the dictionary, a basic data structure that we have not used much, except for very simply in the decision tree (Chapter 12) that consists of a set of keys and values. For a graph, the key to each dictionary entry is the name of the node, and its value is a list of the nodes that it is connected to, as in this example:

```
graph = {'A': ['B', 'C'],'B': ['C', 'D'],'C': ['D'],'D': ['C'],'E': ['F'],
'F': ['C']}
```

That is all there is to it for creating the dictionary, and using it is not very different, since there are built-in methods to get a list of keys (`keys()`) and check if a key is in a dictionary (`in`). Code to find a path through the graph can then be written as a simple recursive function:

```
def findPath(graph, start, end, pathSoFar):
    pathSoFar = pathSoFar + [start]
    if start == end:
        return pathSoFar
    if start not in graph:
        return None
    for node in graph[start]:
        if node not in pathSoFar:
            newpath = findPath(graph, node, end, pathSoFar)
            return newpath
    return None
```

Using those methods we can now look at a Python implementation of the decision tree, which also has a recursive function call as its basis.

12.2.4 Implementation of the Decision Tree

The `make_tree()` function (which uses the `calc_entropy()` and `calc_info_gain()` functions that were described previously) looks like:

```
def make_tree(data,classes,featureNames):
    # Various initialisations suppressed
```

```
default = classes[np.argmax(frequency)]
if nData==0 or nFeatures == 0:
    # Have reached an empty branch
    return default
elif classes.count(classes[0]) == nData:
    # Only 1 class remains
    return classes[0]
else:
    # Choose which feature is best
    gain = np.zeros(nFeatures)
    for feature in range(nFeatures):
        g = calc_info_gain(data,classes,feature)
        gain[feature] = totalEntropy - g
    bestFeature = np.argmax(gain)
    tree = {featureNames[bestFeature]:{}}
    # Find the possible feature values
    for value in values:
        # Find the datapoints with each feature value
        for datapoint in data:
            if datapoint[bestFeature]==value:
                if bestFeature==0:
                    datapoint = datapoint[1:]
                    newNames = featureNames[1:]
                elif bestFeature==nFeatures:
                    datapoint = datapoint[:-1]
                    newNames = featureNames[:-1]
                else:
                    datapoint = datapoint[:bestFeature]
                    datapoint.extend(datapoint[bestFeature+1:])
                    newNames = featureNames[:bestFeature]
                    newNames.extend(featureNames[bestFeature+1:])
                newData.append(datapoint)
                newClasses.append(classes[index])
            index += 1
        # Now recurse to the next level
        subtree = make_tree(newData,newClasses,newNames)
        # And on returning, add the subtree on to the tree
        tree[featureNames[bestFeature]][value] = subtree
    return tree
```

It is worth considering how ID3 generalises from training examples to the set of all possible inputs. It uses a method known as the inductive bias. The choice of the next feature to add into the tree is the one with the highest information gain, which biases the algorithm towards smaller trees, since it tries to minimise the amount of information that is left. This is consistent with a well-known principle that short solutions are usually better than longer ones (not necessarily true, but simpler explanations are usually easier to remember and understand). You might have heard of this principle as 'Occam's Razor', although I prefer

it as an acronym: KISS (Keep It Simple, Stupid). In fact, there is a sound information-theoretic way to write down this principle. It is known as the Minimum Description Length (MDL) and was proposed by Rissanen in 1989. In essence it says that the shortest description of something, i.e., the most compressed one, is the best description.

Note that the algorithm can deal with noise in the dataset, because the labels are assigned to the most common value of the target attribute. Another benefit of decision trees is that they can deal with missing data. Think what would happen if an example has a missing feature. In that case, we can skip that node of the tree and carry on without it, summing over all the possible values that that feature could have taken. This is virtually impossible to do with neural networks: how do you represent missing data when the computation is based on whether or not a neuron is firing? In the case of neural networks it is common to either throw away any datapoints that have missing data, or guess (more technically impute any missing values, either by identifying similar datapoints and using their value or by using the mean or median of the data values for that feature). This assumes that the data that is missing is randomly distributed within the dataset, not missing because of some unknown process.

Saying that ID3 is biased towards short trees is only partly true. The algorithm uses all of the features that are given to it, even if some of them are not necessary. This obviously runs the risk of overfitting, indeed it makes it very likely. There are a few things that you can do to avoid overfitting, the simplest one being to limit the size of the tree. You can also use a variant of early stopping by using a validation set and measuring the performance of the tree so far against it. However, the approach that is used in more advanced algorithms (most notably C4.5, which Quinlan invented to improve on ID3) is pruning.

There are a few versions of pruning, all of which are based on computing the full tree and reducing it, evaluating the error on a validation set. The most naïve version runs the decision tree algorithm until all of the features are used, so that it is probably overfitted, and then produces smaller trees by running over the tree, picking each node in turn, and replacing the subtree beneath every node with a leaf labelled with the most common classification of the subtree. The error of the pruned tree is evaluated on the validation set, and the pruned tree is kept if the error is the same as or less than the original tree, and rejected otherwise.

C4.5 uses a different method called rule post-pruning. This consists of taking the tree generated by ID3, converting it to a set of if-then rules, and then pruning each rule by removing preconditions if the accuracy of the rule increases without it. The rules are then sorted according to their accuracy on the training set and applied in order. The advantages of dealing with rules are that they are easier to read and their order in the tree does not matter, just their accuracy in the classification.

12.2.5 Dealing with Continuous Variables

One thing that we have not yet discussed is how to deal with continuous variables, we have only considered those with discrete sets of feature values. The simplest solution is to discretise the continuous variable. However, it is also possible to leave it continuous and modify the algorithm. For a continuous variable there is not just one place to split it: the variable can be broken between any pair of datapoints, as shown in Figure 12.3. It can, of course, be split in any of the infinite locations along the line as well, but they are no different to this smaller set of locations. Even this smaller set makes the algorithm more expensive for continuous variables than it is for discrete ones, since as well as calculating the information gain of each variable to pick the best one, the information gain of many points within each variable has to be computed. In general, only one split is made to a continuous

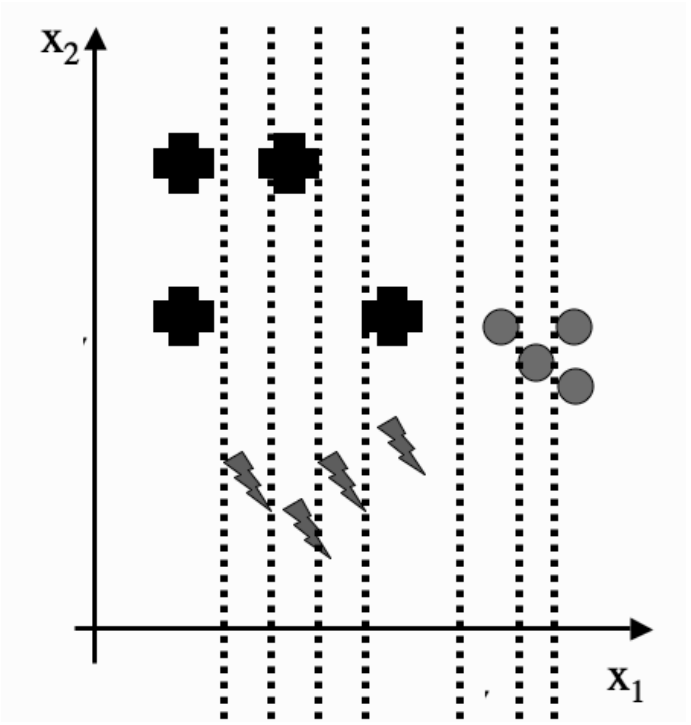

FIGURE 12.3 Possible places to split the variable x_1, between each of the datapoints as the feature value increases.

variable, rather than allowing for threeway or higher splits, although these can be done if necessary.

The trees that these algorithms make are all univariate trees, because they pick one feature (dimension) at a time and split according to that one. There are also algorithms that make multivariate trees by picking combinations of features. This can make for considerably smaller trees if it is possible to find straight lines that separate the data well, but are not parallel to any axis. However, univariate trees are simpler and tend to get good results, so we won't consider multivariate trees any further. This fact that one feature is chosen at a time provides another useful way to visualise what the decision tree is doing. Figure 12.4 shows the idea. Given a dataset that contains three classes, the algorithm picks a feature and value for that feature to split the remaining data into two. The final tree that results from this is shown in Figure 12.5.

12.2.6 Computational Complexity

The computational cost of constructing binary trees is well known for the general case, being $\mathcal{O}(N \log N)$ for construction and $\mathcal{O}(\log N)$ for returning a particular leaf, where N is the number of nodes. However, these results are for balanced binary trees, and decision trees are often not balanced; while the information measures attempt to keep the tree balanced by finding splits that separate the data into two even parts (since that will have the largest entropy), there is no guarantee of this. Nor are they necessarily binary, especially for ID3 and C4.5, as our example shows.

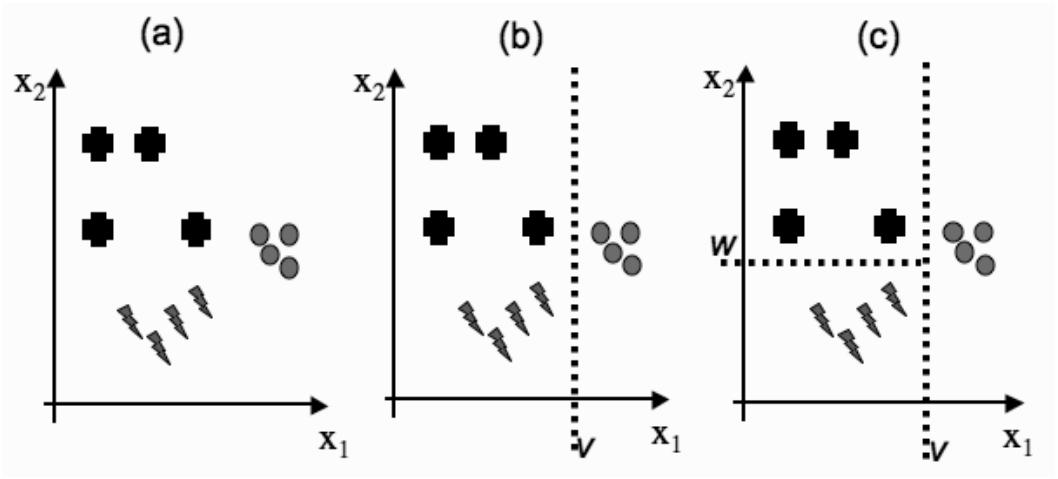

FIGURE 12.4 The effect of decision tree choices. The two-dimensional dataset shown in (a) is split first by choosing feature x_1 (b) and then x_2, (c) which separates out the three classes. The final tree is shown in Figure 12.5.

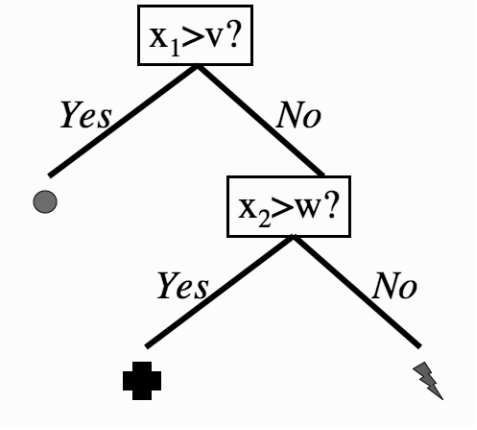

FIGURE 12.5 The final tree created by the splits in Figure 12.4.

If we assume that the tree is approximately balanced, then the cost at each node consists of searching through the d possible features (although this decreases by 1 at each level, that doesn't affect the complexity in the $\mathcal{O}(\cdot)$ notation) and then computing the information gain for the dataset for each split. This has cost $\mathcal{O}(dn \log n)$, where n is the size of the dataset at that node. For the root, $n = N$, and if the tree is balanced, then n is divided by 2 at each stage down the tree. Summing this over the approximately $\log N$ levels in the tree gives computational cost $\mathcal{O}(dN^2 \log N)$.

12.3 CLASSIFICATION AND REGRESSION TREES (CART)

There is another well-known tree-based algorithm, CART, whose name indicates that it can be used for both classification and regression. Classification is not wildly different in CART, although it is usually constrained to construct binary trees. This might seem odd at first, but there are sound computer science reasons why binary trees are good, as suggested in the computational cost discussion above, and it is not a real limation. Even in the example that we started the chapter with, we can always turn questions into binary decisions by splitting the question up a little. Thus, a question that has three answers (say the question about when your nearest assignment deadline is, which is either 'urgent', 'near', or 'none') can be split into two questions: first, 'is the deadline urgent?', and then if the answer to that is 'no', second 'is the deadline near?' The only real difference with classification in CART is that a different information measure is commonly used. This is discussed next, before we look briefly at regression with trees.

12.3.1 Gini Impurity

The entropy that was used in ID3 as the information measure is not the only way to pick features. Another possibility is something known as the Gini impurity. The 'impurity' in the name suggests that the aim of the decision tree is to have each leaf node represent a set of datapoints that are in the same class, so that there are no mismatches. This is known as purity. If a leaf is pure then all of the training data within it have just one class. In which case, if we count the number of datapoints at the node (or better, the fraction of the number of datapoints) that belong to a class i (call it $N(i)$), then it should be 0 for all except one value of i. So suppose that you want to decide on which feature to choose for a split. The algorithm loops over the different features and checks how many points belong to each class. If the node is pure, then $N(i) = 0$ for all values of i except one particular one. So for any particular feature k you can compute:

$$G_k = \sum_{i=1}^{c} \sum_{j \neq i} N(i)N(j), \tag{12.8}$$

where c is the number of classes. In fact, you can reduce the algorithmic effort required by noticing that $\sum_i N(i) = 1$ (since there has to be some output class) and so $\sum_{j \neq i} N(j) = 1 - N(i)$. Then Equation (12.8) is equivalent to:

$$G_k = 1 - \sum_{i=1}^{c} N(i)^2. \tag{12.9}$$

Either way, the Gini impurity is equivalent to computing the expected error rate if the classification was picked according to the class distribution. The information gain can then be measured in the same way, subtracting each value G_i from the total Gini impurity.

The information measure can be changed in another way, which is to add a weight to the misclassifications. The idea is to consider the cost of misclassifying an instance of class i as class j (which we will call the risk in Section 2.3.1) and add a weight that says how important each datapoint is. It is typically labelled as λ_{ij} and is presented as a matrix, with element λ_{ij} representing the cost of misclassifying i as j. Using it is simple, modifying the Gini impurity (Equation (12.8)) to be:

$$G_i = \sum_{j \neq i} \lambda_{ij} N(i) N(j). \tag{12.10}$$

We will see in Section 13.1 that there is another benefit to using these weights, which is to successively improve the classification ability by putting higher weight on datapoints that the algorithm is getting wrong.

12.3.2 Regression in Trees

The new part about CART is its application in regression. While it might seem strange to use trees for regression, it turns out to require only a simple modification to the algorithm. Suppose that the outputs are continuous, so that a regression model is appropriate. None of the node impurity measures that we have considered so far will work. Instead, we'll go back to our old favourite—the sum-of-squares error. To evaluate the choice of which feature to use next, we also need to find the value at which to split the dataset according to that feature. Remember that the output is a value at each leaf. In general, this is just a constant value for the output, computed as the mean average of all the datapoints that are situated in that leaf. This is the optimal choice in order to minimise the sum-of-squares error, but it also means that we can choose the split point quickly for a given feature, by choosing it to minimise the sum-of-squares error. We can then pick the feature that has the split point that provides the best sum-of-squares error, and continue to use the algorithm as for classification.

12.4 CLASSIFICATION EXAMPLE

We'll work through an example using ID3 in this section. The data that we'll use will be a continuation of the one we started the chapter with, about what to do in the evening.

When we want to construct the decision tree to decide what to do in the evening, we start by listing everything that we've done for the past few days to get a suitable dataset (here, the last ten days):

Deadline?	Is there a party?	Lazy?	Activity
Urgent	Yes	Yes	Party
Urgent	No	Yes	Study
Near	Yes	Yes	Party
None	Yes	No	Party
None	No	Yes	Pub
None	Yes	No	Party
Near	No	No	Study
Near	No	Yes	TV
Near	Yes	Yes	Party
Urgent	No	No	Study

To produce a decision tree for this problem, the first thing that we need to do is work out which feature to use as the root node. We start by computing the entropy of S:

$$\begin{aligned}
\text{Entropy}(S) &= -p_{\text{party}} \log_2 p_{\text{party}} - p_{\text{study}} \log_2 p_{\text{study}} \\
&- p_{\text{pub}} \log_2 p_{\text{pub}} - p_{\text{TV}} \log_2 p_{\text{TV}} \\
&= -\frac{5}{10} \log_2 \frac{5}{10} - \frac{3}{10} \log_2 \frac{3}{10} - \frac{1}{10} \log_2 \frac{1}{10} - \frac{1}{10} \log_2 \frac{1}{10} \\
&= 0.5 + 0.5211 + 0.3322 + 0.3322 = 1.6855
\end{aligned} \tag{12.11}$$

and then find which feature has the maximal information gain:

$$\begin{aligned}
\text{Gain}(S, \text{Deadline}) &= 1.6855 - \frac{|S_{\text{urgent}}|}{10} \text{Entropy}(S_{\text{urgent}}) \\
&- \frac{|S_{\text{near}}|}{10} \text{Entropy}(S_{\text{near}}) - \frac{|S_{\text{none}}|}{10} \text{Entropy}(S_{\text{none}}) \\
&= 1.6855 - \frac{3}{10} \left(-\frac{2}{3} \log_2 \frac{2}{3} - \frac{1}{3} \log_2 \frac{1}{3} \right) \\
&- \frac{4}{10} \left(-\frac{2}{4} \log_2 \frac{2}{4} - \frac{1}{4} \log_2 \frac{1}{4} - \frac{1}{4} \log_2 \frac{1}{4} \right) \\
&- \frac{3}{10} \left(-\frac{1}{3} \log_2 \frac{1}{3} - \frac{2}{3} \log_2 \frac{2}{3} \right) \\
&= 1.6855 - 0.2755 - 0.6 - 0.2755 \\
&= 0.5345
\end{aligned} \tag{12.12}$$

$$\begin{aligned}
\text{Gain}(S, \text{Party}) &= 1.6855 - \frac{5}{10} \left(-\frac{5}{5} \log_2 \frac{5}{5} \right) \\
&- \frac{5}{10} \left(-\frac{3}{5} \log_2 \frac{3}{5} - \frac{1}{5} \log_2 \frac{1}{5} - \frac{1}{5} \log_2 \frac{1}{5} \right) \\
&= 1.6855 - 0 - 0.6855 \\
&= 1.0
\end{aligned} \tag{12.13}$$

$$\begin{aligned}
\text{Gain}(S, \text{Lazy}) &= 1.6855 - \frac{6}{10} \left(-\frac{3}{6} \log_2 \frac{3}{6} - \frac{1}{6} \log_2 \frac{1}{6} - \frac{1}{6} \log_2 \frac{1}{6} - \frac{1}{6} \log_2 \frac{1}{6} \right) \\
&- \frac{4}{10} \left(-\frac{2}{4} \log_2 \frac{2}{4} - \frac{2}{4} \log_2 \frac{2}{4} \right) \\
&= 1.6855 - 1.0755 - 0.4 \\
&= 0.21
\end{aligned} \tag{12.14}$$

Therefore, the root node will be the party feature, which has two feature values ('yes' and 'no'), so it will have two branches coming out of it (see Figure 12.6). When we look at the 'yes' branch, we see that in all five cases where there was a party we went to it, so we just put a leaf node there, saying 'party'. For the 'no' branch, out of the five cases there are three different outcomes, so now we need to choose another feature. The five cases we are looking at are:

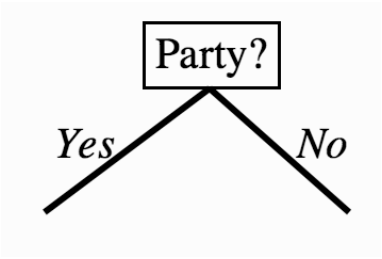

FIGURE 12.6 The decision tree after one step of the algorithm.

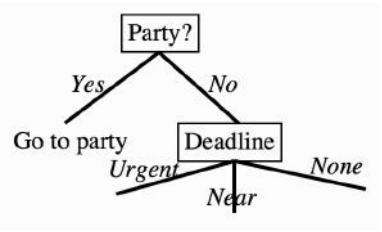

FIGURE 12.7 The tree after another step.

Deadline?	Is there a party?	Lazy?	Activity
Urgent	No	Yes	Study
None	No	Yes	Pub
Near	No	No	Study
Near	No	Yes	TV
Urgent	No	Yes	Study

We've used the party feature, so we just need to calculate the information gain of the other two over these five examples:

$$\begin{aligned}
\text{Gain}(S, \text{Deadline}) &= 1.371 - \frac{2}{5}\left(-\frac{2}{2}\log_2\frac{2}{2}\right) \\
&- \frac{2}{5}\left(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}\right) - \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 1.371 - 0 - 0.4 - 0 \\
&= 0.971 \qquad (12.15) \\
\text{Gain}(S, \text{Lazy}) &= 1.371 - \frac{4}{5}\left(-\frac{2}{4}\log_2\frac{2}{4} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{4}\log_2\frac{1}{4}\right) \\
&- \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 1.371 - 1.2 - 0 \\
&= 0.1710 \qquad (12.16)
\end{aligned}$$

This leads to the tree shown in Figure 12.7. From this point it is relatively simple to complete the tree, leading to the one that was shown in Figure 12.1.

FURTHER READING

For more information about decision trees, the following two books are of interest:

- J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Francisco, CA, USA, 1993.
- L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone. *Classification and Regression Trees.* Chapman & Hall, New York, USA, 1993.

If you want to know more about information theory, then there are lots of books on the topic, including:

- T.M. Cover and J.A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, USA, 1991.
- F.M. Reza. *An Introduction to Information Theory*. McGraw-Hill, New York, USA, 1961.

The original paper that started the field is:

- C.E. Shannon. A mathematical theory of information. *The Bell System Technical Journal*, 27(3):379–423 and 623–656, 1948.

A book that covers information theory and machine learning is:

- D.J.C. MacKay. *Information Thoery, Inference and Learning Algorithms*. Cambridge University Press, Cambridge, UK, 2003.

Other machine learning textbooks that cover decision trees include:

- Sections 8.2–8.4 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.
- Chapter 7 of B.D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, Cambridge, UK, 1996.
- Chapter 3 of T. Mitchell. *Machine Learning*. McGraw-Hill, New York, USA, 1997.

PRACTICE QUESTIONS

Problem 12.1 Suppose that the probability of five events are P(first) = 0.5, and P(second) = P(third) = P(fourth) = P(fifth) = 0.125. Calculate the entropy. Write down in words what this means.

Problem 12.2 Make a decision tree that computes the logical AND function. How does it compare to the Perceptron solution?

Problem 12.3 Turn this politically incorrect data from Quinlan into a decision tree to classify which attributes make a person attractive, and then extract the rules.

Height	Hair	Eyes	Attractive?
Small	Blonde	Brown	No
Tall	Dark	Brown	No
Tall	Blonde	Blue	Yes
Tall	Dark	Blue	No
Small	Dark	Blue	No
Tall	Red	Blue	Yes
Tall	Blonde	Brown	No
Small	Blonde	Blue	Yes

Problem 12.4 When you arrive at the pub, your five friends already have their drinks on the table. Jim has a job and buys the round half of the time. Jane buys the round a quarter of the time, and Sarah and Simon buy a round one eighth of the time. John hasn't got his wallet out since you met him three years ago.

Compute the entropy of each of them buying the round and work out how many questions you need to ask (on average) to find out who bought the round.

Two more friends now arrive and everybody spontaneously decides that it is your turn to buy a round (for all eight of you). Your friends set you the challenge of deciding who is drinking beer and who is drinking vodka according to their gender, whether or not they are students, and whether they went to the pub last night. Use ID3 to work it out, and then see if you can prune the tree.

Drink	Gender	Student	Pub last night
Beer	T	T	T
Beer	T	F	T
Vodka	T	F	F
Vodka	T	F	F
Vodka	F	T	T
Vodka	F	F	F
Vodka	F	T	T
Vodka	F	T	T

Problem 12.5 Use the naïve Bayes classifier from Section 2.3.2 on the datasets that you used for the decision tree (this will involve some effort in turning the textual data into probabilities) and compare the results.

Problem 12.6 The `CPU` dataset in the UCI repository is a very good regression problem for a decision tree. You will need to modify the decision tree code so that it does regression, as discussed in Section 12.3.2. You will also have to work out the Gini impurity for multiple classes.

Problem 12.7 Modify the implementation to deal with continuous variables, as discussed in Section 12.2.5.

Problem 12.8 The misclassification impurity is:

$$N(i) = 1 - \max_j P(w_j). \tag{12.17}$$

Add this into the code and test the new version on some of the datasets above.

Decision by Committee: Ensemble Learning

The old saying has it that two heads are better than one. Which naturally leads to the idea that even more heads are better than that, and ends up with decision by committee, which is famously useless for human activities (as in the old joke that a camel is a horse designed by a committee). For machine learning methods the results are rather more impressive, as we'll see in this chapter.

The basic idea is that by having lots of learners that each get slightly different results on a dataset—some learning certain things well and some learning others—and putting them together, the results that are generated will be significantly better than any one of them on its own (provided that you put them together well... otherwise the results could be significantly worse). One analogy that might prove useful is to think about how your doctor goes about performing a diagnosis of some complaint that you visit her with. If she cannot find the problem directly, then she will ask for a variety of tests to be performed, e.g., scans, blood tests, consultations with experts. She will then aggregate all of these opinions in order to perform a diagnosis. Each of the individual tests will suggest a diagnosis, but only by putting them together can an informed decision be reached.

Figure 13.1 shows the basic idea of ensemble learning, as these methods are collectively called. Given a relatively simple binary classification problem and some learner that puts an ellipse around a subset of the data, combining the ellipses can provide a considerably more complex decision boundary.

There are then only a couple of questions to ask: which learners should we use, how should we ensure that they learn different things, and how should we combine their results? The methods that we are investigating in this chapter can use any classifier at all. Although in general they only use one type of classifier at a time, they do not have to. A common choice of classifier is the decision tree (see Chapter 12).

Ensuring that the learners see different things can be performed in different ways, and it is the primary difference between the algorithms that we shall see. However, it can also come about naturally depending upon the application area. Suppose that you have lots and lots of data. In that case you could simply randomly partition the data and give different sets of data to different classifiers. Even here there are choices: do you make the partitions separate, or include overlaps? If there is no overlap, then it could be difficult to work out how to combine the classifiers, or it might be very simple: if your doctor always asks for opinions from two colleagues, one specialising in heart problems and one in sports injuries,

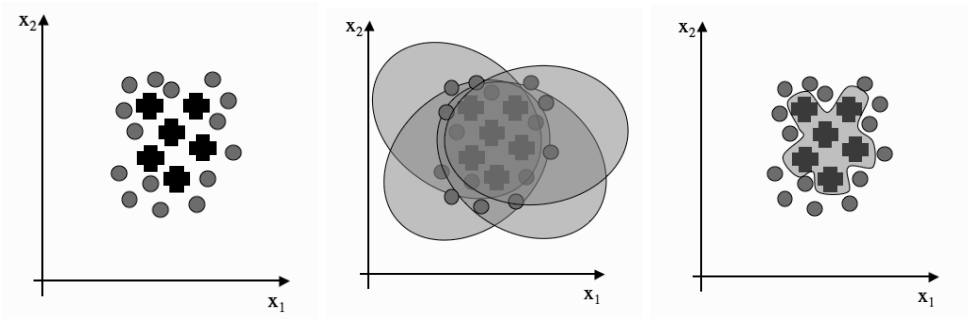

FIGURE 13.1 By combining lots of simple classifiers (here that simply put an elliptical decision boundary onto the data), the decision boundary can be made much more complicated, enabling the difficult separation of the pluses from the circles.

then upon discovering that your leg started hurting after you went for a run she would likely accord more weight to the diagnosis of the sports injury expert.

Interestingly, ensemble methods do very well when there is very little data as well as when there is too much. To see why, think cross-validation (Section 2.2.2). We used cross-validation when there was not enough data to go around, and trained lots of neural networks on different subsets of the data. Then we threw away most of them. With an ensemble method we keep them all, and combine their results in some way. One very simple way to combine the results is to use majority voting — if it's good enough for electing governments in elections, it's good enough for machine learning. Majority voting has the interesting property that for binary classification, the combined classifier will only get the answer wrong if more than half of the classifiers were wrong. Hopefully, this isn't going to happen too often (although you might be able to think of government elections where this has been the case in your view). There are alternative ways to combine the results, as we'll discuss. These things will become clearer as we look at the algorithms, so let's get started.

13.1 BOOSTING

At first sight the claim of the most popular ensemble method, boosting, seems amazing. If we take a collection of very poor (weak in the jargon) learners, each performing only just better than chance, then by putting them together it is possible to make an ensemble learner that can perform arbitrarily well. So we just need lots of low-quality learners, and a way to put them together usefully, and we can make a learner that will do very well.

The principal algorithm of boosting is named AdaBoost, and is described in Section 13.1.1. The algorithm was first described in the mid-1990s by Freund and Shapiro, and while it has had many variations derived from it, the principal algorithm is still one of the most widely used. The algorithm was proposed as an improvement on the original 1990 boosting algorithm, which was rather data hungry. In that algorithm, the training set was split into three. A classifier was trained on the first third, and then tested on the second third. All of the data that was misclassified during that testing was used to form a new dataset, along with an equally sized random selection of the data that was correctly classified. A second classifier was trained on this new dataset, and then both of the classifiers were tested on the final third of the dataset. If they both produced the same output, then that datapoint was ignored, otherwise the datapoint was added to yet another new

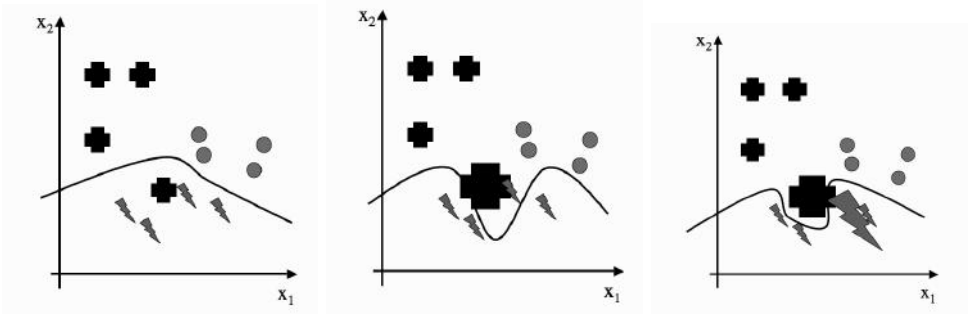

FIGURE 13.2 As points are misclassified, so their weights increase in boosting (shown by the datapoint getting larger), which makes the importance of those datapoints increase, making the classifiers pay more attention to them.

dataset, which formed the training set for a third classifer. Rather than looking further at this version, we will look at the more common algorithm.

13.1.1 AdaBoost

The innovation that AdaBoost (which stands for adaptive boosting) uses is to give weights to each datapoint according to how difficult previous classifiers have found to get it correct. These weights are given to the classifier as part of the input when it is trained.

The AdaBoost algorithm is conceptually very simple. At each iteration a new classifier is trained on the training set, with the weights that are applied to the training set for each datapoint being modified at each iteration according to how successfully that datapoint has been classified in the past. The weights are initially all set to the same value, $1/N$, where N is the number of datapoints in the training set. Then, at each iteration, the error (ϵ) is computed as the sum of the weights of the misclassified points, and the weights for incorrect examples are updated by being multiplied by $\alpha = (1 - \epsilon)/\epsilon$. Weights for correct examples are left alone, and then the whole set is normalised so that it sums to 1 (which is effectively a reduction in the importance of the correctly classified datapoints). Training terminates after a set number of iterations, or when either all of the datapoints are classified correctly, or one point contains more than half of the available weight.

Figure 13.2 shows the effect of weighting incorrectly classified examples as training proceeds, with the size of each datapoint being a measure of its importance. As an algorithm this looks like (where $I(y_n \neq h_t(x_n))$ is an indicator function that returns 1 if the target and output are not equal, and 0 if they are):

AdaBoost Algorithm

- Initialise all weights to $1/N$, where N is the number of datapoints
- While $0 < \epsilon_t < \frac{1}{2}$ (and $t < T$, some maximum number of iterations):
  - train classifier on $\{S, w^{(t)}\}$, getting hypotheses $h_t(x_n)$ for datapoints x_n
  - compute training error $\epsilon_t = \sum_{n=1}^{N} w_n^{(t)} I(y_n \neq h_t(x_n))$
  - set $\alpha_t = \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
  - update weights using:

$$w_n^{(t+1)} = w_n^{(t)} \exp(\alpha_t I(y_n \neq h_t(x_n))/Z_t, \tag{13.1}$$

    where Z_t is a normalisation constant
- Output $f(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$

There is nothing too difficult to the implementation, either, as can be seen from the main loop here:

```
for t in range(T):
    classifiers[:,t] = train(data,classes,w[:,t])
    outputs,errors = classify(data,classifiers[0,t],classifiers[1,t])

    index[:,t] = errors
    print "index: ", index[:,t]
    e[t] = np.sum(w[:,t]*index[:,t])/np.sum(w[:,t])

    if t>0 and (e[t]==0 or e[t]>=0.5):
        T=t
        alpha = alpha[:t]
        index = index[:,:t]
        w = w[:,:t]
        break

    alpha[t] = np.log((1-e[t])/e[t])
    w[:,t+1] = w[:,t]* np.exp(alpha[t]*index[:,t])
    w[:,t+1] = w[:,t+1]/np.sum(w[:,t+1])
```

Most of the work of the algorithm is done by the classification algorithm, which is given new weights at each iteration. In this respect, boosting is not quite a stand-alone algorithm: the classifiers need to consider the weights when they perform their classifications. It is not always obvious how to do this for a particular classifier, but we have seen methods of doing it for a few classifiers. For the decision tree we saw a method in Section 12.3.1,

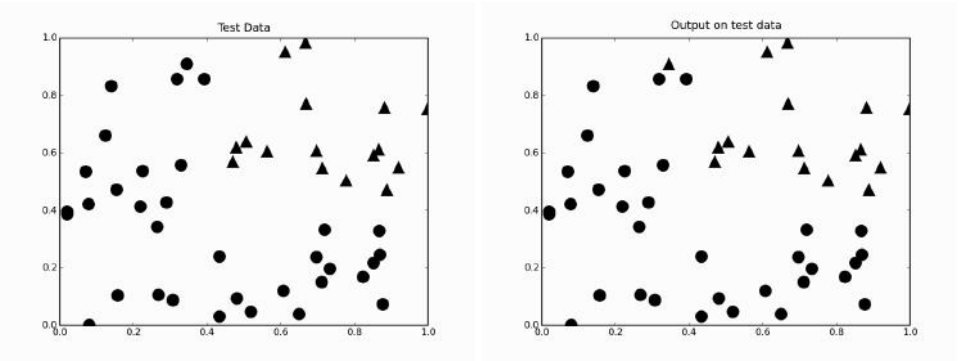

FIGURE 13.3 Boosting learns this simple dataset very successfully, producing an ensemble classifier that is rather more complicated than the simple horizontal or vertical line classifier that the algorithm boosts. On the independent test set shown here, the algorithm gets only 1 datapoint wrong, and that is one that is coincidentally close to one that was misclassified to simulate noise in the training data.

when we looked at the Gini impurity. There, we allowed for a $\boldsymbol{\lambda}$ matrix that encoded the risks associated with misclassification, and these are a perfect place in which to introduce weights. Modification of the decision tree algorithm to deal with these weights is suggested as an exercise for this chapter. A similar argument can be used for the Bayes' classifier; this was discussed in Section 2.3.1.

As a very simple example showing how boosting works, a very simple classifier was created that can only separate data by fitting one either horizontal or vertical line, with it choosing which to fit at the current iteration at random. A two-dimensional dataset was created with data in the top right-hand corner being in one class, and the rest in another, plus a couple of the datapoints were randomly mislabelled to simulate noise. Clearly, this dataset cannot be separated by a single horizontal or vertical decision boundary. However, Figure 13.3 shows the output of the classifier on an independent test set, where the algorithm gets only one datapoint wrong, and that is one that is coincidentally close to one of the 'noisy' datapoints in the training data. Figure 13.4 shows the training data, the error curve on both the training and testing sets, and the first few iterations of the classifier, which can only put in one horizontal or linear classification line.

Clearly, such impressive results require some explanation and understanding. The key to this understanding is to compute the loss function, which is simply the measure of the error that is applied (we have been using a sum-of-squares loss function for many algorithms in the book). The loss function for AdaBoost has the form

$$G_t(\alpha) = \sum_{n=1}^{N} \exp\left(-y_n(\alpha h_t(x_n) + f_{t-1}(x_n))\right), \tag{13.2}$$

where $f_{t-1}(x_n)$ is the sum of the hypotheses of that datapoint from the previous iterations:

$$f_{t-1}(x_n) = \sum_{\tau=0}^{t-1} \alpha_\tau h_\tau(x_n). \tag{13.3}$$

Exponential loss functions are well behaved and robust to outliers. The weights $w^{(t)}$

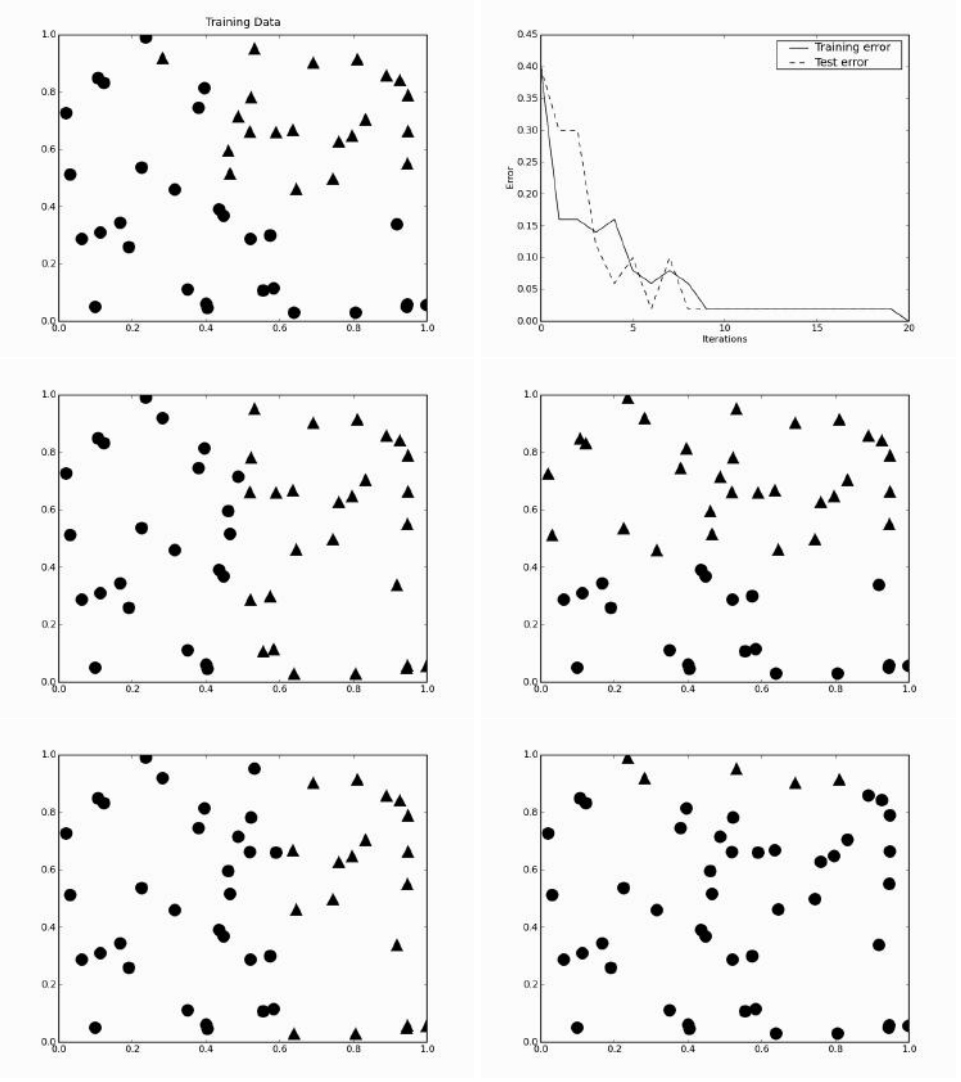

FIGURE 13.4 *Top:* the training data and the error curve. *Middle and bottom:* The first few iterations of the classifier; each plot shows the output of one of the weak classifiers that are boosted by the algorithm.

in the algorithm are nothing more than the second term in Equation (13.2), which can therefore be rewritten as:

$$G_t(\alpha) = \sum_{n=1}^{N} w^{(t)} \exp\left(-y_n \alpha h_t(x_n)\right). \tag{13.4}$$

Deriving the rest of the algorithm from here requires substituting in for the hypotheses h and then solving for α, which produces the full algorithm. Interestingly, this is not the way that AdaBoost was created; this understanding of why it works so well came later. It is possible to choose other loss functions, and providing that they are differentiable they will provide useful boosting-like algorithms, which are collectively known as arcing algorithms (for adaptive reweighting and combining).

AdaBoost can be modified to perform regression rather than classification (known as real adaboost, or sometimes adaboost.R). There is another variant on boosting (also called AdaBoost, confusingly) that uses the weights to sample from the full dataset, training on a sample of the data rather than the full weighted set, with more difficult examples more likely to be in the training sample. This is more in line with the original boosting algorithm, and is obviously faster, since each training run has fewer data to learn about.

13.1.2 Stumping

There is a very extreme form of boosting that is applied to trees. It goes by the descriptive name of stumping. The stump of a tree is the tiny piece that is left over when you chop off the rest, and the same is true here: stumping consists of simply taking the root of the tree and using that as the decision maker. So for each classifier you use the very first question that makes up the root of the tree, and that is it. Often, this is worse than chance on the whole dataset, but by using the weights to sort out when that classifier should be used, and to what extent, as opposed to the other ones, the overall output of stumping can be very successful. In fact, it is pretty much exactly what the simple example that we saw consisted of.

13.2 BAGGING

The simplest method of combining classifiers is known as bagging, which stands for bootstrap aggregating, the statistical description of the method. This is fine if you know what a bootstrap is, but fairly useless if you don't. A bootstrap sample is a sample taken from the original dataset with replacement, so that we may get some data several times and others not at all. The bootstrap sample is the same size as the original, and lots and lots of these samples are taken: B of them, where B is at least 50, and could even be in the thousands. The name bootstrap is more popular in computer science than anywhere else, since there is also a bootstrap loader, which is the first program to run when a computer is turned on. It comes from the nonsensical idea of 'picking yourself up by your bootstraps,' which means lifting yourself up by your shoelaces, and is meant to imply starting from nothing.

Bootstrap sampling seems like a very strange thing to do. We've taken a perfectly good dataset, mucked it up by sampling from it, which might be good if we had made a smaller dataset (since it would be faster), but we still ended up with a dataset the same size. Worse, we've done it lots of times. Surely this is just a way to burn up computer time without gaining anything. The benefit of it is that we will get lots of learners that perform slightly differently, which is exactly what we want for an ensemble method. Another benefit is that estimates of the accuracy of the classification function can be made without complicated

analytic work, by throwing computer resources at the problem (technically, bagging is a variance reducing algorithm; the meaning of this will become clearer when we talk about bias and variance in Section 2.5). This is a standard technique in modern statistics; we'll see another example in Chapter 15 when we look at Markov Chain Monte Carlo methods. It is sufficiently common to have inspired the comment that "statistics is defined as the discipline where those that think don't count and those that count don't think."

Having taken a set of bootstrap samples, the bagging method simply requires that we fit a model to each dataset, and then combine them by taking the output to be the majority vote of all the classifiers. A NumPy implementation is shown next, and then we will look at a simple example.

```
# Compute bootstrap samples
samplePoints = np.random.randint(0,nPoints,(nPoints,nSamples))
classifiers = []

for i in range(nSamples):
 sample = []
 sampleTarget = []
 for j in range(nPoints):
  sample.append(data[samplePoints[j,i]])
  sampleTarget.append(targets[samplePoints[j,i]])
 # Train classifiers
 classifiers.append(self.tree.make_tree(sample,sampleTarget,features))
```

The example consists of taking the party data that was used in Section 12.4 to demonstrate the decision tree, and restricting the trees to stumps, so that they can make a classification based on just one variable. The output of a decision tree that uses the whole dataset for this is not surprising: it takes the two largest classes, and separates them. However, using just stumps of trees and 20 samples, bagging can separate the data perfectly, as this output shows:

```
Tree Stump Prediction
['Party', 'Party', 'Party', 'Party', 'Pub', 'Party', 'Study', 'Study', ↩
'Party', 'Study']
Correct Classes
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', ↩
'Study']
Bagged Results
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', ↩
'Study']
```

13.2.1 Subagging

For some reason, ensemble methods often have good names, such as boosting and bagging (and we will see my choice for best-named, bragging, in Section 13.4). However, the method of subagging wins the prize for the oddest sounding word. It is a combination of 'subsample'

and 'bagging,' and it is the fairly obvious idea that you don't need to produce samples that are the same size as the original data. If you make smaller datasets, then it makes sense to sample without replacement, but otherwise the implementation is only very slightly different from the bagging one, except that in NumPy you use `np.random.shuffle()` to produce the samples. It is common to use a dataset size that is half that of the original data, and the results of this can often be comparable to a full bagging simulation.

13.3 RANDOM FORESTS

If there is one method in machine learning that has grown in popularity over the last few years, then it is the idea of random forests. The concept has been around for longer than that, with several different people inventing variations, but the name that is most strongly attached to it is that of Breiman, who also described the CART algorithm that was discussed in Section 12.2, and also gave bagging its name.

The idea is largely that if one tree is good, then many trees (a forest) should be better, provided that there is enough variety between them. The most interesting thing about a random forest is the ways that it creates randomness from a standard dataset. The first of the methods that it uses is the one that we have just seen: bagging. If we wish to create a forest then we can make the trees different by training them on slightly different data, so we take bootstrap samples from the dataset for each tree. However, this isn't enough randomness yet. The other obvious place where it is possible to add randomness is to limit the choices that the decision tree can make. At each node, a random subset of the features is given to the tree, and it can only pick from that subset rather than from the whole set.

As well as increasing the randomness in the training of each tree, it also speeds up the training, since there are fewer features to search over at each stage. Of course, it does introduce a new parameter (how many features to consider), but the random forest does not seem to be very sensitive to this parameter; in practice, a subset size that is the square root of the number of features seems to be common. The effect of these two forms of randomness is to reduce the variance without effecting the bias. Another benefit of this is that there is no need to prune the trees. There is another parameter that we don't know how to choose yet, which is the number of trees to put into the forest. However, this is fairly easy to pick if we want optimal results: we can keep on building trees until the error stops decreasing.

Once the set of trees are trained, the output of the forest is the majority vote for classification, as with the other committee methods that we have seen, or the mean response for regression. And those are pretty much the main features needed for creating a random forest. The algorithm is given next before we see some results of using the random forest.

The Basic Random Forest Training Algorithm

- For each of N trees:
  - create a new bootstrap sample of the training set
  - use this bootstrap sample to train a decision tree
  - at each node of the decision tree, randomly select m features, and compute the information gain (or Gini impurity) only on that set of features, selecting the optimal one
  - repeat until the tree is complete

The implementation of this is very easy: we modify the decision to take an extra parameter, which is m, the number of features that should be used in the selection set at each stage. We will look at an example of using it shortly as a comparison to boosting.

Looking at the algorithm you might be able to see that it is a very unusual machine learning method because it is embarrassingly parallel: since the trees do not depend upon each other, you can both create and get decisions from different trees on different individual processors if you have them. This means that the random forest can run on as many processors as you have available with nearly linear speedup.

There is one more nice thing to mention about random forests, which is that with a little bit of programming effort they come with built-in test data: the bootstrap sample will miss out about 35% of the data on average, the so-called out-of-bootstrap examples. If we keep track of these datapoints then they can be used as novel samples for that particular tree, giving an estimated test error that we get without having to use any extra datapoints. This avoids the need for cross-validation.

As a brief example of using the random forest, we start by demonstrating that the random forest gets the correct results on the Party example that has been used in both this and the previous chapters, based on 10 trees, each trained on 7 samples, and with just two levels allowed in each tree:

```
RF prediction
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', ↵
'Study']
```

As a rather more involved example, the car evaluation dataset in the UCI Repository contains 1,728 examples aiming to classify whether or not a car is a good purchase based on six attributes. The following results compare a single decision tree, bagging, and a random forest with 50 trees, each based on 100 samples, and with a maximum depth of five for each tree. It can be seen that the random forest is the most accurate of the three methods.

```
Tree
Number correctly predicted 777.0
Number of testpoints  864
Percentage Accuracy  89.9305555556

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 18.0
Percentage Accuracy 46.1538461538
-----
Bagger
Number correctly predicted 678.0
Number of testpoints  864
Percentage Accuracy  78.4722222222

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 0.0
Percentage Accuracy 0.0
-----
```

```
Forest
Number correctly predicted 793.0
Number of testpoints  864
Percentage Accuracy  91.7824074074

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 20.0
Percentage Accuracy 51.28205128
```

13.3.1 Comparison with Boosting

There are some obvious similarities to boosting (Section 13.1), but it is the differences that are most telling. The most general thing is that boosting is exhaustive, in that it searches over the whole set of features at each stage, and each stage depends on the previous one. This means that boosting has to run sequentially, and the individual steps can be expensive to run. By way of contrast, the parallelism of the random forest and the fact that it only searches over a fairly small set of features at each stage speed the algorithm up a lot.

Since the algorithm only searches a small subset of the data at each stage, it cannot be expected to be as good as boosting for the same number of trees. However, since the trees are cheaper to train, we can make more of them in the same computational time, and often the results are amazingly good even on very large and complicated datasets.

In fact, the most amazing thing about random forests is that they seem to deal very well with really big datasets. It is fairly clear that they should do well computationally, since both the reduced number of features to search over and the ability to parallelise should help there. However, they seem to also produce good outputs based on surprisingly small parts of the problem space seen by each tree.

13.4 DIFFERENT WAYS TO COMBINE CLASSIFIERS

Bagging puts most of its effort into ensuring that the different classifiers see different data, since they see different samples of the data. This is different than boosting, where the data stays the same, but the importance of each datapoint changes for the different classifiers, since they each get different weights according to how well the previous classifiers have performed. Just as important for an ensemble method, though, is how it combines the outputs of the different classifiers. Both boosting and bagging take a vote from amongst the classifiers, although they do it in different ways: boosting takes a weighted vote, while bagging simple takes the majority vote. There are other alternatives to these methods, as well.

In fact, even majority voting is not necessarily simple. Some classification systems will only produce an output where all the classifiers agree, or more than half of them agree, whereas others simply take the most common output, which is what we usually mean by majority voting. The idea of not always producing an output is to ensure that the ensemble does not produce outputs that are contentious, because they are probably difficult datapoints. If the number of classifiers is odd and the classifiers are each independent of each other, then majority voting will return the correct label if more than half of the classifiers agree. Assuming that each individual classifier has a success rate of p, the probability of the ensemble getting the correct answer is a binomial distribution of the form:

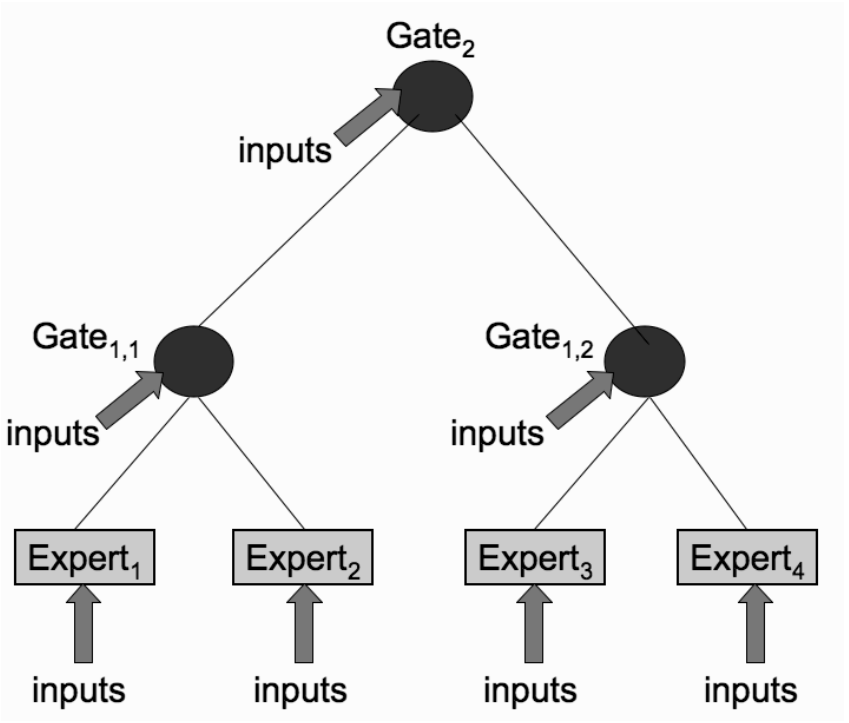

FIGURE 13.5 The Hierarchical Mixture of Networks network, consisting of a set of classifiers (experts) with gating systems that also use the inputs to decide which classifiers to trust.

$$\sum_{k=T/2+1}^{T} \left(\begin{array}{c} T \\ k \end{array} \right) p^k (1-p)^{T-k}, \tag{13.5}$$

where T is the number of classifiers. If $p > 0.5$, then this sum approaches 1 as $T \to \infty$. This is a lot of the power behind ensemble methods: even if each classifier only gets about half the answers right, if we use a decent number of classifiers (maybe 100), then the probability of the ensemble being correct gets close to 1. In fact, even with less than 50% chance of success for each individual classifier, the ensemble can often do very well indeed.

For regression problems, rather than taking the majority vote, it is common to take the mean of the outputs. However, the mean is heavily affected by outliers, with the result that the median is a more common average to use. It is the use of the median that produces the bragging algorithm, which is meant to imply 'robust bagging'.

There is one more thing that can be done to combine classifiers, and that is to learn how to do it. There is an algorithm that does precisely this, known as the mixture of experts. Inputs are presented to the network, and each individual classifier makes an assessment. These outputs from the classifiers are then weighted by the relevant gate, which produces a weight w using the current inputs, and this is propagated further up the hierarchy. The most common version of the mixture of experts works as follows:

The Mixture of Experts Algorithm

- For each expert:
  - calculate the probability of the input belonging to each possible class by computing (where the $\mathbf{w}_i$ are the weights for that classifier):

$$o_i(\mathbf{x}, \mathbf{w}_i) = \frac{1}{1 + \exp(-\mathbf{w}_i \cdot \mathbf{x})}. \tag{13.6}$$

- For each gating network up the tree:
  - compute:

$$g_i(\mathbf{x}, \mathbf{v}_i) = \frac{\exp(\mathbf{v}_i \mathbf{x})}{\sum_l \exp(\mathbf{v}_l \mathbf{x})}. \tag{13.7}$$

- Pass as input to the next level gates (where the sum is over the relevant inputs to that gate):

$$\sum_k o_j g_j. \tag{13.8}$$

The most common way to train this network is using an EM algorithm. This is a general statistical approximation algorithm that is discussed in Section 7.1.1. It is also possible to use gradient descent on the parameters.

There are a couple of other ways to view these mixture of experts methods. One is to regard them as trees, except that the splits are not the hard splits that we performed in Chapter 12, but rather soft, because they are based on probability. The other is to compare them with radial basis function (RBF) networks (see Section 5.2). Each RBF gave a constant output within its receptive field. If, instead, each node were to give a linear approximation to the data, then the result would be the mixture of experts network.

FURTHER READING

Three papers that cover the three main ensemble methods described in this section are:

- R.E. Schapire. The boosting approach to machine learning: An overview. In D. D. Denison, M. H. Hansen, C. Holmes, B. Mallick, and B. Yu, editors, *Nonlinear Estimation and Classification*, Springer, Berlin, Germany, 2003.
- L. Breiman. Bagging predictors. *Machine Learning*, 26(2):123–140, 1996.
- M.I. Jordan and R.A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6(2):181–214, 1994.

An overview of the whole area is provided by:

- L. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms.* Wiley-Interscience, New York, USA, 2004.

For an alternative viewpoint, see:

- Sections 17.4 and 17.6–17.7 of E. Alpaydin. *Introduction to Machine Learning*, 2nd edition, MIT Press, Cambridge, MA, USA, 2009.

- Section 9.5 of R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*, 2nd edition, Wiley-Interscience, New York, USA, 2001.

The original paper on Random Forests is still a very useful resource:
Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

PRACTICE QUESTIONS

Problem 13.1 Modify the decision tree implementation to use weights in the computation of the Gini impurity. This is not trivial, since you have to modify the total value of the Gini impurity, too. Once you have done it, use stump trees on the party data.

Problem 13.2 Implement the alternative form of boosting that uses the weights to sample the dataset. Does this make any difference to the outputs?

Problem 13.3 Stumping picks out the single most informative feature in the dataset and uses this. For a binary classification problem this will typically get at least half of the dataset correct. Why? How does this statement generalise to multiple classes?

Problem 13.4 Compare and contrast bagging and cross-validation.

Problem 13.5 The `Breastcancer` dataset in the UCI Machine Learning repository gives ten features and asks for a classification of breast tumours into benign and malignant. It is a difficult dataset, and provides a good comparison of the standard decision tree with boosted and bagged versions. Use all of the methods, using stumping and more advanced trees and see which work better.

Problem 13.6 The Mixture of Experts algorithm works with any kind of expert. Suppose that the experts were each MLPs. Implement this algorithm and see how well it does on the `Breastcancer` dataset above.

Problem 13.7 In Section 13.3 on the random forest, it was mentioned that there exists out-of-bootstrap data that can be used for validation and testing. Modify the code to keep track of this data.

Problem 13.8 Use the boosting code from Problem 13.2 above and compare it with the random forest on the cars dataset from the UCI Repository.