CHAPTER 7

Modeling with Decision Trees

You've now seen a few different automatic classifiers, and this chapter will expand on them by introducing a very useful method called *decision tree learning*. Unlike most other classifiers, the models produced by decision trees are easy to interpret—the list of numbers in a Bayesian classifier will tell you how important each word is, but you really have to do the calculation to know what the outcome will be. A neural network is even more difficult to interpret, since the weight of the connection between two neurons has very little meaning on its own. You can understand the reasoning process of a decision tree just by looking at it, and you can even convert it to a simple series of if-then statements.

This chapter will cover three different examples that employ decision trees. The first shows how to predict which of a site's users are likely to pay for premium access. Many online applications that are priced by subscription or on a per-use basis offer users a way to try the applications before spending money. In the case of subscriptions, the sites usually offer a time-limited free trial or a feature-limited free version. Sites that employ per-use pricing may offer a free session or similar incentive.

The other examples, covered later in the chapter, will use decision trees to model housing prices and "hotness."

Predicting Signups

Sometimes when a high-traffic site links to a new application that offers free accounts and subscription accounts, the application will get thousands of new users. Many of these users are driven by curiosity and are not really looking for that particular type of application, so there is a very low likelihood that they will become paying customers. This makes it difficult to distinguish and follow up with likely customers, so many sites resort to mass-emailing everyone who has signed up, rather than using a more targeted approach.

To help with this problem, it would be useful to be able to predict the likelihood that a user will become a paying customer. You know by now that you can use a Bayesian classifier or neural network to do this. However, clarity is very important in this case—if you know the factors that indicate a user will become a customer, you can use that information to guide an advertising strategy, to make certain aspects of the site more accessible, or to use other strategies that will help increase the number of paying customers.

For this example, imagine an online application that offers a free trial. Users sign up for the trial and use the site for a certain number of days, after which they can choose to upgrade to a basic or premium service. As users sign up for the free trial, information about them is collected, and at the end of the trial, the site owners note which users chose to become paying customers.

To minimize annoyance for users and sign them up as quickly as possible, the site doesn't ask them a lot of questions about themselves—instead, it collects information from the server logs, such as the site that referred them, their geographical location, how many pages they viewed before signing up, and so on. If you collect the data and put it in a table, it might look like Table 7-1.

Table 7-1. User behavior and final purchase decision for a web site

Referrer	Location	Read FAQ	Pages viewed	Service chosen
Slashdot	USA	Yes	18	None
Google	France	Yes	23	Premium
Digg	USA	Yes	24	Basic
Kiwitobes	France	Yes	23	Basic
Google	UK	No	21	Premium
(direct)	New Zealand	No	12	None
(direct)	UK	No	21	Basic
Google	USA	No	24	Premium
Slashdot	France	Yes	19	None
Digg	USA	No	18	None
Google	UK	No	18	None
Kiwitobes	UK	No	19	None
Digg	New Zealand	Yes	12	Basic
Google	UK	Yes	18	Basic
Kiwitobes	France	Yes	19	Basic

Arrange the data in a list of rows, with each row being a list of columns. The final column in each row indicates whether or not the user signed up; this Service column is the value you want to be able to predict. Create a new file called *treepredict.py* to work with throughout this chapter. If you'd like to enter the data manually, add this to the top of the file:

```
my_data=[['slashdot','USA','yes',18,'None'],
        ['google','France','yes',23,'Premium'],
        ['digg','USA','yes',24,'Basic'],
        ['kiwitobes','France','yes',23,'Basic'],
        ['google','UK','no',21,'Premium'],
        ['(direct)','New Zealand','no',12,'None'],
        ['(direct)','UK','no',21,'Basic'],
        ['google','USA','no',24,'Premium'],
        ['slashdot','France','yes',19,'None'],
        ['digg','USA','no',18,'None'],
        ['google','UK','no',18,'None'],
        ['kiwitobes','UK','no',19,'None'],
        ['digg','New Zealand','yes',12,'Basic'],
        ['slashdot','UK','no',21,'None'],
        ['google','UK','yes',18,'Basic'],
        ['kiwitobes','France','yes',19,'Basic']]
```

If you'd prefer to download the dataset, it's available at *http://kiwitobes.com/tree/decision_tree_example.txt*.

To load in the file, add this line to the top of *treepredict.py*:

```
my_data=[line.split('\t') for line in file('decision_tree_example.txt')]
```

You now have information about users' locations, where they connected from, and how much time they spent on your site before signing up; you just need a way to fill in the Service column with a good guess.

Introducing Decision Trees

Decision trees are one of the simpler machine-learning methods. They are a completely transparent method of classifying observations, which, after training, look like a series of if-then statements arranged into a tree. Figure 7-1 shows an example of a decision tree for classifying fruit.

Once you have a decision tree, it's quite easy to see how it makes all of its decisions. Just follow the path down the tree that answers each question correctly and you'll eventually arrive at an answer. Tracing back from the node where you ended up gives a rationale for the final classification.

This chapter will look at a way to represent a decision tree, at code for constructing the tree from real data, and at code for classifying new observations. The first step is to create a representation of a tree. Create a new class called decisionnode, which represents each node in the tree:

```
class decisionnode:
  def __init__(self,col=-1,value=None,results=None,tb=None,fb=None):
    self.col=col
    self.value=value
    self.results=results
    self.tb=tb
    self.fb=fb
```

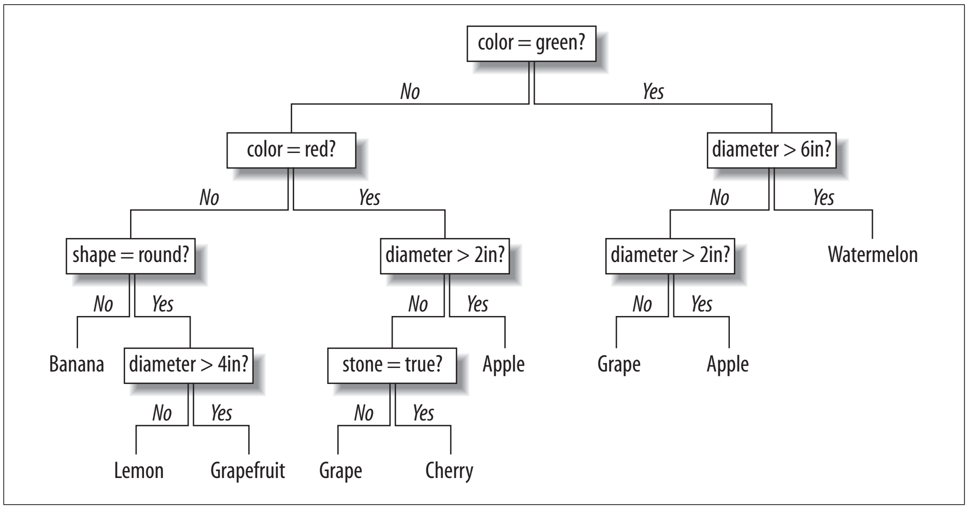


Figure 7-1. Example decision tree

Each node has five instance variables, all of which may be set in the initializer:

- `col` is the column index of the criteria to be tested.
- `value` is the value that the column must match to get a true result.
- `tb` and `fb` are `decisionnodes`, which are the next nodes in the tree if the result is true or false, respectively.
- `results` stores a dictionary of results for this branch. This is `None` for everything except endpoints.

The functions that create a tree return the root node, which can be traversed by following its True or False branches until a branch with results is reached.

Training the Tree

This chapter uses an algorithm called *CART* (Classification and Regression Trees). To build the decision tree, the algorithm first creates a root node. By considering all the observations in the table, it chooses the best variable to divide up the data. To do this, it looks at all the different variables and decides which condition (for example, "Did the user read the FAQ?") would separate the outcomes (which service the user signed up for) in a way that makes it easier to guess what the user will do.

`divideset` is a function that divides the rows into two sets based on the data in a specific column. This function takes a list of rows, a column number, and a value to divide into the column. In the case of Read FAQ, the possible values are Yes or No, and for Referrer, there are several possibilities. It then returns two lists of rows: the first containing the rows where the data in the specified column matches the value, and the second containing the rows where it does not.

```
# Divides a set on a specific column. Can handle numeric
# or nominal values
def divideset(rows,column,value):
   # Make a function that tells us if a row is in
   # the first group (true) or the second group (false)
   split_function=None
   if isinstance(value,int) or isinstance(value,float):
      split_function=lambda row:row[column]>=value
   else:
      split_function=lambda row:row[column]==value

   # Divide the rows into two sets and return them
   set1=[row for row in rows if split_function(row)]
   set2=[row for row in rows if not split_function(row)]
   return (set1,set2)
```

The code creates a function to divide the data called `split_function`, which depends on knowing if the data is numerical or not. If it is, the `true` criterion is that the value in this column is greater than `value`. If the data is not numeric, `split_function` simply determines whether the column's value is the same as `value`. It uses this function to divide the data into two sets, one where `split_function` returns `true` and one where it returns `false`.

Start a Python session and try dividing the results by the Read FAQ column:

```
$ python
>>> import treepredict
>>> treepredict.divideset(treepredict.my_data,2,'yes')
([['slashdot', 'USA', 'yes', 18, 'None'], ['google', 'France', 'yes', 23,
'Premium'],...]]
[['google', 'UK', 'no', 21, 'Premium'], ['(direct)', 'New Zealand', 'no', 12,
'None'],...])
```

Table 7-2 shows the division.

Table 7-2. Outcomes based on Read FAQ column values

True	False
None	Premium
Premium	None
Basic	Basic
Basic	Premium
None	None
Basic	None
Basic	None

This doesn't look like a good variable for separating the outcomes at this stage, since both sides seem pretty well mixed. We need a way to choose the best variable.

Choosing the Best Split

Our casual observation that the chosen variable isn't very good may be accurate, but to choose which variable to use in a software solution, you need a way to measure how mixed a set is. What you want to do is find the variable that creates the two sets with the least possible mixing. The first function you'll need is one to get the counts of each result in a set. Add this to *treepredict.py*:

```
# Create counts of possible results (the last column of
# each row is the result)
def uniquecounts(rows):
   results={}
   for row in rows:
      # The result is the last column
      r=row[len(row)-1]
      if r not in results: results[r]=0
      results[r]+=1
   return results
```

`uniquecounts` finds all the different possible outcomes and returns them as a dictionary of how many times they each appear. This is used by the other functions to calculate how mixed a set is. There are a few different metrics for measuring this, and two will be considered here: Gini impurity and entropy.

Gini Impurity

Gini impurity is the expected error rate if one of the results from a set is randomly applied to one of the items in the set. If every item in the set is in the same category, the guess will always be correct, so the error rate is 0. If there are four possible results evenly divided in the group, there's a 75 percent chance that the guess would be incorrect, so the error rate is 0.75.

The function for Gini impurity looks like this:

```
# Probability that a randomly placed item will
# be in the wrong category
def giniimpurity(rows):
  total=len(rows)
  counts=uniquecounts(rows)
  imp=0
  for k1 in counts:
    p1=float(counts[k1])/total
    for k2 in counts:
      if k1==k2: continue
      p2=float(counts[k2])/total
      imp+=p1*p2
  return imp
```

This function calculates the probability of each possible outcome by dividing the number of times that outcome occurs by the total number of rows in the set. It then adds up the products of all these probabilities. This gives the overall chance that a row would be randomly assigned to the wrong outcome. The higher this probability, the worse the split. A probability of zero is great because it tells you that everything is already in the right set.

Entropy

Entropy, in information theory, is the amount of disorder in a set—basically, how mixed a set is. Add this function to *treepredict.py*:

```
# Entropy is the sum of p(x)log(p(x)) across all
# the different possible results
def entropy(rows):
   from math import log
   log2=lambda x:log(x)/log(2)
   results=uniquecounts(rows)
   # Now calculate the entropy
   ent=0.0
   for r in results.keys( ):
      p=float(results[r])/len(rows)
      ent=ent-p*log2(p)
   return ent
```

The entropy function calculates the frequency of each item (the number of times it appears divided by the total number of rows), and applies these formulas:

p(i) = frequency(outcome) = count(outcome) / count(total rows)
Entropy = sum of p(i) x log(p(i)) for all outcomes

This is a measurement of how different the outcomes are from each other. If they're all the same (e.g., if you were really lucky and everyone became a premium subscriber), then the entropy is 0. The more mixed up the groups are, the higher their entropy. Our goal in dividing the data into two new groups is to reduce the entropy.

Try testing the Gini impurity and entropy metrics in your Python session:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.py'>
>>> treepredict.giniimpurity(treepredict.my_data)
0.6328125
>>> treepredict.entropy(treepredict.my_data)
1.5052408149441479
>>> set1,set2=treepredict.divideset(treepredict.my_data,2,'yes')
>>> treepredict.entropy(set1)
1.2987949406953985
>>> treepredict.giniimpurity(set1)
0.53125
```

The main difference between entropy and Gini impurity is that entropy peaks more slowly. Consequently, it tends to penalize mixed sets a little more heavily. The rest of this chapter will use entropy as the metric because it is used more commonly, but it's easy to swap it out for the Gini impurity.

Recursive Tree Building

To see how good an attribute is, the algorithm first calculates the entropy of the whole group. Then it tries dividing up the group by the possible values of each attribute and calculates the entropy of the two new groups. To determine which attribute is the best to divide on, the *information gain* is calculated. Information gain is the difference between the current entropy and the weighted-average entropy of the two new groups. The algorithm calculates the information gain for every attribute and chooses the one with the highest information gain.

After the condition for the root node has been decided, the algorithm creates two branches corresponding to true or false for that condition, as shown in Figure 7-2.

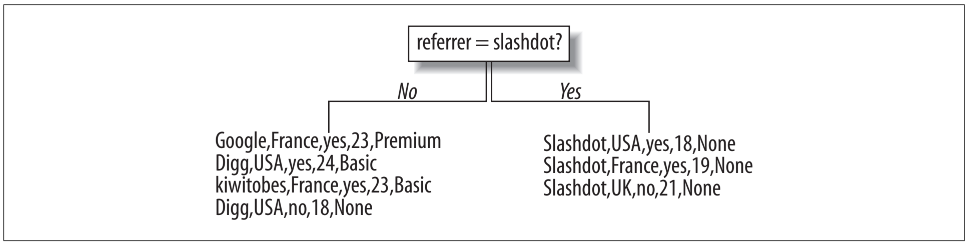


Figure 7-2. Decision tree after a single split

The observations are divided into those that meet the condition and those that don't. For each branch, the algorithm then determines if the branch can be divided further or if it has reached a solid conclusion. If one of the new branches can be divided, the same method as above is used to determine which variable to use. The second division is shown in Figure 7-3.

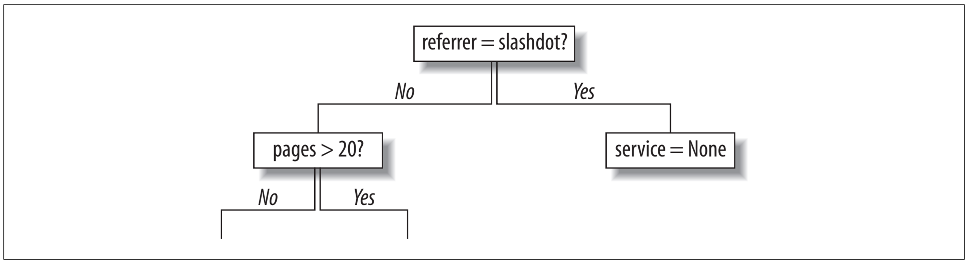


Figure 7-3. Decision tree after two splits

The branches keep dividing, creating a tree by calculating the best attribute for each new node. A branch stops dividing when the information gain from splitting a node is not more than zero.

Create a new function called `buildtree` in *treepredict.py*. This is a recursive function that builds the tree by choosing the best dividing criteria for the current set:

```
def buildtree(rows,scoref=entropy):
  if len(rows)==0: return decisionnode( )
  current_score=scoref(rows)

  # Set up some variables to track the best criteria
  best_gain=0.0
  best_criteria=None
  best_sets=None

  column_count=len(rows[0])-1
  for col in range(0,column_count):
    # Generate the list of different values in
    # this column
    column_values={}
    for row in rows:
       column_values[row[col]]=1
    # Now try dividing the rows up for each value
    # in this column
    for value in column_values.keys( ):
      (set1,set2)=divideset(rows,col,value)

      # Information gain
      p=float(len(set1))/len(rows)
      gain=current_score-p*scoref(set1)-(1-p)*scoref(set2)
      if gain>best_gain and len(set1)>0 and len(set2)>0:
        best_gain=gain
        best_criteria=(col,value)
        best_sets=(set1,set2)
  # Create the subbranches
  if best_gain>0:
    trueBranch=buildtree(best_sets[0])
    falseBranch=buildtree(best_sets[1])
    return decisionnode(col=best_criteria[0],value=best_criteria[1],
                        tb=trueBranch,fb=falseBranch)
  else:
    return decisionnode(results=uniquecounts(rows))
```

This function is first called with the list of rows. It loops through every column (except the last one, which has the result in it), finds every possible value for that column, and divides the dataset into two new subsets. It calculates the weighted-average entropy for every pair of new subsets by multiplying each set's entropy by the fraction of the items that ended up in each set, and remembers which pair has the lowest entropy.

If the best pair of subsets doesn't have a lower weighted-average entropy than the current set, that branch ends and the counts of the possible outcomes are stored. Otherwise, `buildtree` is called on each set and they are added to the tree. The results of the calls on each subset are attached to the True and False branches of the nodes, eventually constructing an entire tree.

Now you can finally apply the algorithm to the original dataset. The code above is flexible enough to handle both text and numeric data. It also assumes that the last row is the target value, so you can simply pass the rows of data to build the tree:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.py'>
>>> tree=treepredict.buildtree(treepredict.my_data)
```

`tree` now holds a trained decision tree. In a moment you'll learn how to look at the tree, and later, how to use it to make predictions.

Displaying the Tree

So now that you have a tree, what should you do with it? Well, one thing you'll definitely want to do is look at it. `printtree` is a simple function for displaying the tree in plain text. The output isn't pretty, but it's a simple way to view small trees:

```
def printtree(tree,indent=''):
   # Is this a leaf node?
   if tree.results!=None:
      print str(tree.results)
   else:
      # Print the criteria
      print str(tree.col)+':'+str(tree.value)+'? '

      # Print the branches
      print indent+'T->',
      printtree(tree.tb,indent+'  ')
      print indent+'F->',
      printtree(tree.fb,indent+'  ')
```

This is another recursive function. It takes a tree returned by `buildtree` and traverses down it, and it knows it has reached the end of a branch when it reaches the node with `results`. Until it reaches that point, it prints the criteria for the True and False branches and calls `printtree` on each of them, each time increasing the indent string.

Call this function with the tree you just built, and you'll get something like this:

```
>>> reload(treepredict)
>>> treepredict.printtree(tree)
0:google?
T-> 3:21?
  T-> {'Premium': 3}
  F-> 2:yes?
    T-> {'Basic': 1}
    F-> {'None': 1}
F-> 0:slashdot?
  T-> {'None': 3}
  F-> 2:yes?
    T-> {'Basic': 4}
    F-> 3:21?
      T-> {'Basic': 1}
      F-> {'None': 3}
```

This is a visual representation of the process that the decision tree will go through when trying to make a new classification. The condition on the root node is “is Google in column 0?” If this condition is met, it proceeds to the `T->` branch and finds that anyone referred from Google will become a paid subscriber if they have viewed 21 pages or more. If the condition is not met, it jumps to the `F->` branch and evaluates the condition “is Slashdot in column 0?” This continues until it reaches a branch that has a result. As mentioned earlier, the ability to view the logic behind the reasoning process is one of the big advantages of decision trees.

Graphical Display

The textual display of the tree is fine for small trees, but as they get larger, visually tracking your way through the tree can be quite difficult. Here you’ll see how to make a graphical representation of the tree that will be useful for viewing trees you’ll build in later sections.

The code for drawing the tree is similar to the code for drawing dendrograms in Chapter 3. Both involve drawing a binary tree with nodes of arbitrary depth, so you’ll first need functions to decide how much space a given node will take up—both the total width of all its children and how deep the node goes, which tells you much vertical space it will need for all its branches. The total width of a branch is the combined width of its child branches, or 1 if it doesn’t have any child branches:

```
def getwidth(tree):
  if tree.tb==None and tree.fb==None: return 1
  return getwidth(tree.tb)+getwidth(tree.fb)
```

The depth of a branch is 1 plus the total depth of its longest child branch:

```
def getdepth(tree):
  if tree.tb==None and tree.fb==None: return 0
  return max(getdepth(tree.tb),getdepth(tree.fb))+1
```

To actually draw the tree, you’ll need to have the Python Imaging Library installed. You can get this library from *http://pythonware.com*, and Appendix A has more information on installing it. Add this `import` statement at the beginning of *treepredict.py*:

```
from PIL import Image,ImageDraw
```

The `drawtree` function determines the appropriate total size and sets up a canvas. It then passes this canvas and the top node of the tree to `drawnode`. Add this function to *treepredict.py*:

```
def drawtree(tree,jpeg='tree.jpg'):
  w=getwidth(tree)*100
  h=getdepth(tree)*100+120

  img=Image.new('RGB',(w,h),(255,255,255))
  draw=ImageDraw.Draw(img)

  drawnode(draw,tree,w/2,20)
  img.save(jpeg,'JPEG')
```

The drawnode function actually draws the decision nodes of the tree. It works recursively, first drawing the current node and calculating the positions of the child nodes, then calling drawnode on each of the child nodes. Add this function to *treepredict.py*:

```
def drawnode(draw,tree,x,y):
  if tree.results==None:
    # Get the width of each branch
    w1=getwidth(tree.fb)*100
    w2=getwidth(tree.tb)*100

    # Determine the total space required by this node
    left=x-(w1+w2)/2
    right=x+(w1+w2)/2

    # Draw the condition string
    draw.text((x-20,y-10),str(tree.col)+':'+str(tree.value),(0,0,0))

    # Draw links to the branches
    draw.line((x,y,left+w1/2,y+100),fill=(255,0,0))
    draw.line((x,y,right-w2/2,y+100),fill=(255,0,0))

    # Draw the branch nodes
    drawnode(draw,tree.fb,left+w1/2,y+100)
    drawnode(draw,tree.tb,right-w2/2,y+100)
  else:
    txt=' \n'.join(['%s:%d'%v for v in tree.results.items( )])
    draw.text((x-20,y),txt,(0,0,0))
```

You can now try drawing the current tree in your Python session:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.pyc'>
>>> treepredict.drawtree(tree,jpeg='treeview.jpg')
```

This should produce a new file called *treeview.jpg*, which is shown in Figure 7-4.

The code does not print the True and False branch labels, and they would likely just contribute to the clutter of larger diagrams. In the generated tree diagrams, the True branch is always the righthand branch, so you can follow the reasoning process through.

Classifying New Observations

Now you'll need a function that takes a new observation and classifies it according to the decision tree. Add this function to *treepredict.py*:

```
def classify(observation,tree):
  if tree.results!=None:
    return tree.results
  else:
    v=observation[tree.col]
    branch=None
    if isinstance(v,int) or isinstance(v,float):
```

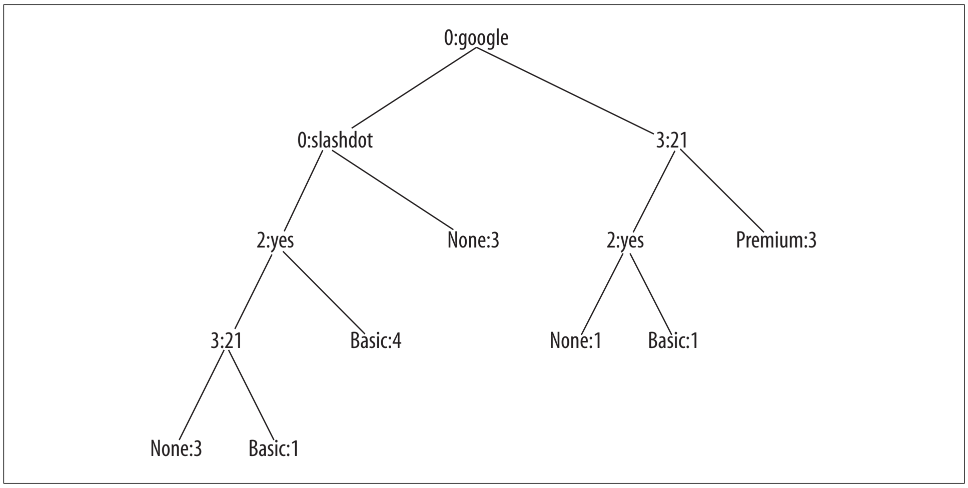


Figure 7-4. Decision tree for predicting subscribers

```
          if v>=tree.value: branch=tree.tb
          else: branch=tree.fb
        else:
          if v==tree.value: branch=tree.tb
          else: branch=tree.fb
        return classify(observation,branch)
```

This function traverses the tree in much the same manner as `printtree`. After each call, it checks to see if it has reached the end of this branch by looking for `results`. If not, it evaluates the observation to see if the column matches the value. If it does, it calls `classify` again on the True branch; if not, it calls `classify` on the False branch.

Now you can call `classify` to get the prediction for a new observation:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.pyc'>
>>> treepredict.classify(['(direct)','USA','yes',5],tree)
{'Basic': 4}
```

You now have functions for creating a decision tree from any dataset, for displaying and interpreting the tree, and for classifying new results. These functions can be applied to any dataset that consists of multiple rows, each containing a set of observations and an outcome.

Pruning the Tree

One problem with training the tree using the methods described so far is that it can become *overfitted*—that is, it can become too specific to the training data. An overfitted tree may give an answer as being more certain than it really is by creating branches that decrease entropy slightly for the training set, but whose conditions are actually completely arbitrary.

Decision Trees in the Real World

Because decision trees are so easy to interpret, they are among the most widely used data-mining methods in business analysis, medical decision-making, and policy-making. Often, a decision tree is created automatically, and an expert uses it to understand the key factors and then refines it to better match her beliefs. This process allows machines to assist experts and to clearly show the reasoning process so that individuals can judge the quality of the prediction.

Decision trees have been used in this manner for such wide-ranging applications as customer profiling, financial risk analysis, assisted diagnosis, and traffic prediction.

Since the algorithm above continually splits the branches until it can't reduce the entropy any further, one possibility is to stop splitting when the entropy is not reduced by a minimum amount. This strategy is employed frequently, but it suffers from a minor drawback—it is possible to have a dataset where the entropy is not reduced much by one split but is reduced greatly by subsequent splits. An alternative strategy is to build the entire tree as described earlier, and then try to eliminate superfluous nodes. This process is known as *pruning*.

Pruning involves checking pairs of nodes that have a common parent to see if merging them would increase the entropy by less than a specified threshold. If so, the leaves are merged into a single node with all the possible outcomes. This helps avoid overfitting and stops the tree from making predictions that are more confident than what can really be gleaned from the data.

Add a new function to *treepredict.py* for pruning the tree:

```
def prune(tree,mingain):
  # If the branches aren't leaves, then prune them
  if tree.tb.results==None:
    prune(tree.tb,mingain)
  if tree.fb.results==None:
    prune(tree.fb,mingain)

  # If both the subbranches are now leaves, see if they
  # should merged
  if tree.tb.results!=None and tree.fb.results!=None:
    # Build a combined dataset
    tb,fb=[],[]
    for v,c in tree.tb.results.items( ):
      tb+=[[v]]*c
    for v,c in tree.fb.results.items( ):
      fb+=[[v]]*c

    # Test the reduction in entropy
    delta=entropy(tb+fb)-(entropy(tb)+entropy(fb)/2)
```

```
        if delta<mingain:
          # Merge the branches
          tree.tb,tree.fb=None,None
          tree.results=uniquecounts(tb+fb)
```

When this function is called on the root node, it will traverse all the way down the tree to the nodes that only have leaf nodes as children. It will create a combined list of results from both of the leaves and will test the entropy. If the change in entropy is less than the `mingain` parameter, the leaves will be deleted and all their results moved to their parent node. The combined node then becomes a possible candidate for deletion and merging with another node.

Try it on your current dataset to see if it merges any of the nodes:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.pyc'>
>>> treepredict.prune(tree,0.1)
>>> treepredict.printtree(tree)
0:google?
T-> 3:21?
  T-> {'Premium': 3}
  F-> 2:yes?
    T-> {'Basic': 1}
    F-> {'None': 1}
F-> 0:slashdot?
  T-> {'None': 3}
  F-> 2:yes?
    T-> {'Basic': 4}
    F-> 3:21?
      T-> {'Basic': 1}
      F-> {'None': 3}
>>> treepredict.prune(tree,1.0)
>>> treepredict.printtree(tree)
0:google?
T-> 3:21?
  T-> {'Premium': 3}
  F-> 2:yes?
    T-> {'Basic': 1}
    F-> {'None': 1}
F-> {'None': 6, 'Basic': 5}
```

In the example, the data divides quite easily, so pruning with a reasonable minimum gain doesn't really do anything. Only when the minimum gain is turned up very high does one of the leaves get merged. As you'll see later, real datasets tend not to break as cleanly as this one does, so pruning is much more effective in those cases.

Dealing with Missing Data

Another advantage of decision trees is their ability to deal with missing data. Your dataset may be missing some piece of information—in the current example, for instance, the geographical location of a user may not be discernable from her IP address, so the field may be blank. To adapt the decision tree to handle this, you'll need to implement a different prediction function.

If you are missing a piece of data that is required to decide which branch of the tree to follow, you can actually follow *both* branches. However, instead of counting the results equally, the results from either side are weighted. In the basic decision tree, everything has an implied weight of 1, meaning that the observations count fully for the probability that an item fits into a certain category. If you are following multiple branches instead, you can give each branch a weight equal to the fraction of all the other rows that are on that side.

The function for doing this, `mdclassify`, is a simple modification of `classify`. Add it to *treepredict.py*:

```
def mdclassify(observation,tree):
  if tree.results!=None:
    return tree.results
  else:
    v=observation[tree.col]
    if v==None:
      tr,fr=mdclassify(observation,tree.tb),mdclassify(observation,tree.fb)
      tcount=sum(tr.values( ))
      fcount=sum(fr.values( ))
      tw=float(tcount)/(tcount+fcount)
      fw=float(fcount)/(tcount+fcount)
      result={}
      for k,v in tr.items( ): result[k]=v*tw
      for k,v in fr.items( ): result[k]=v*fw
      return result
    else:
      if isinstance(v,int) or isinstance(v,float):
        if v>=tree.value: branch=tree.tb
        else: branch=tree.fb
      else:
        if v==tree.value: branch=tree.tb
        else: branch=tree.fb
      return mdclassify(observation,branch)
```

The only difference is at the end where, if the important piece of data is missing, the results for each branch are calculated and then combined with their respective weightings.

Try out `mdclassify` on a row with a crucial piece of information missing and see how your results look:

```
>>> reload(treepredict)
<module 'treepredict' from 'treepredict.py'>
>>> treepredict.mdclassify(['google',None,'yes',None],tree)
{'Premium': 1.5, 'Basic': 1.5}
>>> treepredict2.mdclassify(['google','France',None,None],tree)
{'None': 0.125, 'Premium': 2.25, 'Basic': 0.125}
```

As expected, leaving out the Pages variable returns a strong chance of Premium and a slight chance of Basic. Leaving out the Read FAQ variable yields a different distribution, with each possibility in the end weighted by how many items were placed on each side.

Dealing with Numerical Outcomes

The user behavior example and the fruit tree were both classification problems, since the outcomes were categories rather than numbers. The remaining examples in this chapter, home prices and hotness, are both problems with numerical outcomes.

While it's possible to run `buildtree` on a dataset with numbers as outcomes, the result probably won't be very good. If all the numbers are treated as different categories, the algorithm won't take into account the fact that some numbers are close together and others are far apart; they will all be treated as completely separate. To deal with this, when you have a tree with numerical outcomes, you can use `variance` as a scoring function instead of entropy or Gini impurity. Add the `variance` function to *treepredict.py*:

```
def variance(rows):
  if len(rows)==0: return 0
  data=[float(row[len(row)-1]) for row in rows]
  mean=sum(data)/len(data)
  variance=sum([(d-mean)**2 for d in data])/len(data)
  return variance
```

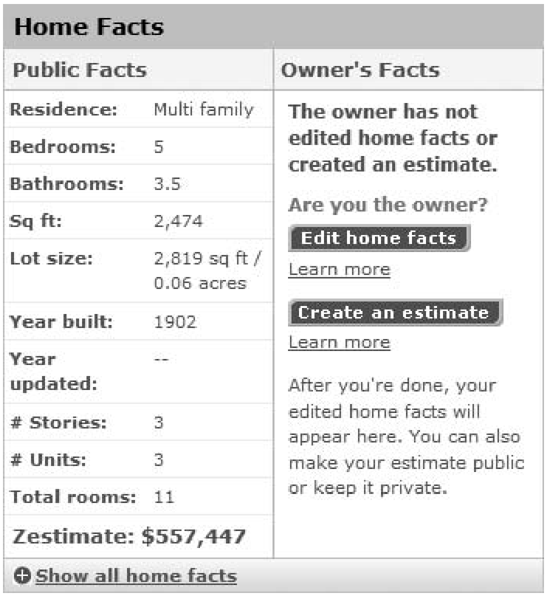
This function is a possible parameter for `buildtree`, and it calculates the statistical variance for a set of rows. A low variance means that the numbers are all very close together, and a high variance means that they are widely dispersed. When building a tree using variance as the scoring function, node criteria will be picked that split the numbers so that higher values are on one side and lower values are on the other. Splitting the data this way reduces the overall variance on the branches.

Modeling Home Prices

There are many potential uses for decision trees, but they are most useful when there are several possible variables and you're interested in the reasoning process. In some cases, you already know the outcomes, and the interesting part is modeling the outcomes to understand why they are as they are. One area in which this is potentially very interesting is understanding prices of goods, particularly those that have a lot of variability in measurable ways. This section will look at building decision trees for modeling real estate prices, because houses vary greatly in price and have many numerical and nominal variables that are easily measured.

The Zillow API

Zillow is a free web service that tracks real estate prices and uses this information to create price estimates for other houses. It works by looking at comps (similar houses) and using their values to predict a new value, which is similar to what real estate appraisers do. A section of a Zillow web page showing information about a house and its estimate value is shown in Figure 7-5.



Home Facts

Public Facts		Owner's Facts
Residence:	Multi family	The owner has not edited home facts or created an estimate.
Bedrooms:	5	
Bathrooms:	3.5	Are you the owner?
Sq ft:	2,474	Edit home facts
Lot size:	2,819 sq ft / 0.06 acres	Learn more
Year built:	1902	Create an estimate
Year updated:	--	Learn more
# Stories:	3	After you're done, your edited home facts will appear here. You can also make your estimate public or keep it private.
# Units:	3	
Total rooms:	11	
Zestimate: $557,447		

Show all home facts

Figure 7-5. Screenshot from zillow.com

Fortunately, Zillow also has an API that lets you get details and the estimated value of houses. The page for the Zillow API is *http://www.zillow.com/howto/api/APIOverview.htm*.

You'll need to get a developer key to access the API, which is free and available from the web site. The API itself is quite simple—it involves requesting a URL with all your search parameters in the query, and then parsing the returned XML to get details like number of bedrooms and estimated price. Create a new file called *zillow.py* and add the following code:

```
import xml.dom.minidom
import urllib2

zwskey="X1-ZWz1chwxis15aj_9skq6"
```

As you did in Chapter 5, you're going to use the *minidom* API to parse XML results of your queries. The function getaddressdata takes an address and a city, and constructs the URL to query Zillow for property information. It parses the results and extracts the important information, which it returns as a tuple of results. Add this function to *zillow.py*:

```
def getaddressdata(address,city):
  escad=address.replace(' ','+')

  # Construct the URL
  url='http://www.zillow.com/webservice/GetDeepSearchResults.htm?'
  url+='zws-id=%s&address=%s&citystatezip=%s' % (zwskey,escad,city)

  # Parse resulting XML
  doc=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))
  code=doc.getElementsByTagName('code')[0].firstChild.data

  # Code 0 means success; otherwise, there was an error
  if code!='0': return None

  # Extract the info about this property
  try:
    zipcode=doc.getElementsByTagName('zipcode')[0].firstChild.data
    use=doc.getElementsByTagName('useCode')[0].firstChild.data
    year=doc.getElementsByTagName('yearBuilt')[0].firstChild.data
    bath=doc.getElementsByTagName('bathrooms')[0].firstChild.data
    bed=doc.getElementsByTagName('bedrooms')[0].firstChild.data
    rooms=doc.getElementsByTagName('totalRooms')[0].firstChild.data
    price=doc.getElementsByTagName('amount')[0].firstChild.data
  except:
    return None

  return (zipcode,use,int(year),float(bath),int(bed),int(rooms),price)
```

The tuple returned by this function is suitable to put in a list as an observation, since the "result," the price bucket, is at the end. To use this function to generate an entire dataset, you'll need a list of addresses. You can generate this yourself or download a list of randomly generated addresses in Cambridge, MA at *http://kiwitobes.com/addresslist.txt*.

Create a new function called `getpricelist` to read this file and generate a list of data:

```
def getpricelist( ):
  l1=[]
  for line in file('addresslist.txt'):
    data=getaddressdata(line.strip( ),'Cambridge,MA')
    l1.append(data)
  return l1
```

You can now use these functions to generate a dataset and build a decision tree. Try this in your Python session:

```
>>> import zillow
>>> housedata=zillow.getpricelist( )
>>> reload(treepredict)
>>> housetree=treepredict.buildtree(housedata,scoref=treepredict.variance)
>>> treepredict.drawtree(housetree,'housetree.jpg')
```

One possible generated file, *housetree.jpg*, is shown in Figure 7-6.

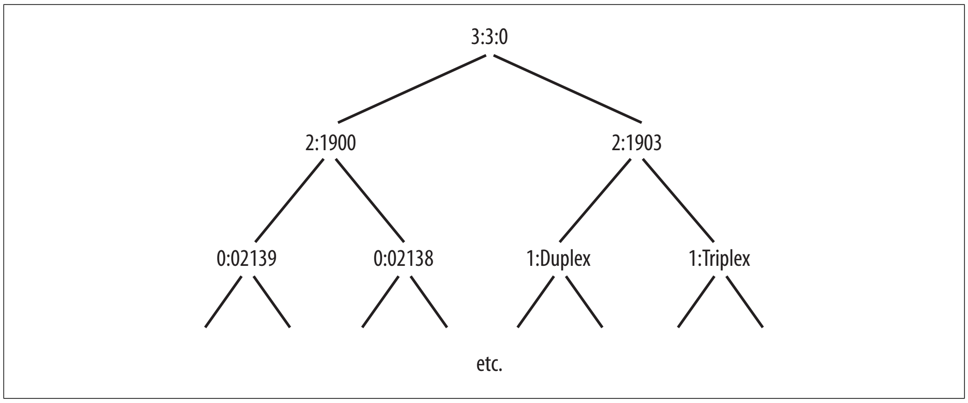


Figure 7-6. Decision tree for house prices

Of course, if you were only interested in guessing the price of particular property, you could just use the Zillow API to get an estimate. What's interesting here is that you actually built a model of the factors to be considered in determining housing prices. Notice that the top of the tree is Bathrooms, which means that you reduce the variance the most by dividing the dataset on the total number of bathrooms. The main deciding factor in the price of a house in Cambridge is whether or not it has three or more bathrooms (usually this indicates that the property is a large multifamily house).

The obvious downside of using a decision tree here is that it's necessary to create buckets of price data, since they're all different and have to be grouped in some way to create useful endpoints. It's possible that a different prediction technique would have worked better on the actual price data. Chapter 8 discusses a different method for making price predictions.

Modeling "Hotness"

Hot or Not is a site that allows users to upload photos of themselves. Its original concept was to let users rank other users on their physical appearance, and to aggregate the results to create a score between 1 and 10 for each person. It has since evolved into a dating site, and now has an open API that allows you to get demographic information about members along with their "hotness" rating. This makes it an interesting test case for a decision tree model because there is a set of input variables, an output variable, and a possibly interesting reasoning process. The site itself is also a good example of what might be considered collective intelligence.

Again, you'll need to get an application key to access the API. You can sign up and get one at *http://dev.hotornot.com/signup*.

The Hot or Not API works in much the same way as the other APIs that have been covered. You simply pass the parameters of a query to a URL and parse the XML that is returned. To get started, create a new file called *hotornot.py* and add the `import` statements and your key definition:

```
import urllib2
import xml.dom.minidom

api_key="479NUNJHETN"
```

Next, get a list of random people to make up the dataset. Fortunately, Hot or Not provides an API call that returns a list of people with specified criteria. In this example, the only criteria will be that the people have "meet me" profiles, since only from these profiles can you get other information like location and interests. Add this function to *hotornot.py*:

```
def getrandomratings(c):
  # Construct URL for getRandomProfile
  url="http://services.hotornot.com/rest/?app_key=%s" % api_key
  url+="&method=Rate.getRandomProfile&retrieve_num=%d" % c
  url+="&get_rate_info=true&meet_users_only=true"

  f1=urllib2.urlopen(url).read( )

  doc=xml.dom.minidom.parseString(f1)

  emids=doc.getElementsByTagName('emid')
  ratings=doc.getElementsByTagName('rating')

  # Combine the emids and ratings together into a list
  result=[]
  for e,r in zip(emids,ratings):
    if r.firstChild!=None:
      result.append((e.firstChild.data,r.firstChild.data))
  return result
```

Once you've generated a list of user IDs and ratings, you'll need another function to download information about people—in this case, gender, age, location, and keywords. Having all 50 states as possible location variables will lead to too many branching possibilities. In order to reduce the number of possibilities for location, you can divide the states into regions. Add the following code to specify regions:

```
stateregions={'New England':['ct','mn','ma','nh','ri','vt'],
              'Mid Atlantic':['de','md','nj','ny','pa'],
              'South':['al','ak','fl','ga','ky','la','ms','mo',
                       'nc','sc','tn','va','wv'],
              'Midwest':['il','in','ia','ks','mi','ne','nd','oh','sd','wi'],
              'West':['ak','ca','co','hi','id','mt','nv','or','ut','wa','wy']}
```

The API provides a method to download demographic data for individuals, so the function `getpeopledata` just loops through all the results of the first search and queries the API for their details. Add this function to *hotornot.py*:

```
def getpeopledata(ratings):
  result=[]
  for emid,rating in ratings:
    # URL for the MeetMe.getProfile method
    url="http://services.hotornot.com/rest/?app_key=%s" % api_key
    url+="&method=MeetMe.getProfile&emid=%s&get_keywords=true" % emid

    # Get all the info about this person
    try:
      rating=int(float(rating)+0.5)
      doc2=xml.dom.minidom.parseString(urllib2.urlopen(url).read( ))
      gender=doc2.getElementsByTagName('gender')[0].firstChild.data
      age=doc2.getElementsByTagName('age')[0].firstChild.data
      loc=doc2.getElementsByTagName('location')[0].firstChild.data[0:2]

      # Convert state to region
      for r,s in stateregions.items( ):
        if loc in s: region=r

      if region!=None:
        result.append((gender,int(age),region,rating))
    except:
      pass
  return result
```

You can now import this module into your Python session and generate a dataset:

```
>>> import hotornot
>>> l1=hotornot.getrandomratings(500)
>>> len(l1)
442
>>> pdata=hotornot.getpeopledata(l1)
>>> pdata[0]
(u'female', 28, 'West', 9)
```

The list contains information about each user with their rating as the last field. This data structure can be passed directly to the buildtree method to build a tree:

```
>>> hottree=treepredict.buildtree(pdata,scoref=treepredict.variance)
>>> treepredict.prune(hottree,0.5)
>>> treepredict.drawtree(hottree,'hottree.jpg')
```

A possible output for the final tree is shown in Figure 7-7.

The central node at the top that divides the dataset the best is gender. The remainder of the tree is actually quite complicated and difficult to read. However, you can certainly use it to make predictions about previously unseen people. Also, because the algorithms support missing data, you can aggregate people across large variables. For example, maybe you want to compare the hotness of everyone in the South against everyone in the Mid-Atlantic:

```
>>> south=treepredict2.mdclassify((None,None,'South'),hottree)
>>> midat=treepredict2.mdclassify((None,None,'Mid Atlantic'),hottree)
```

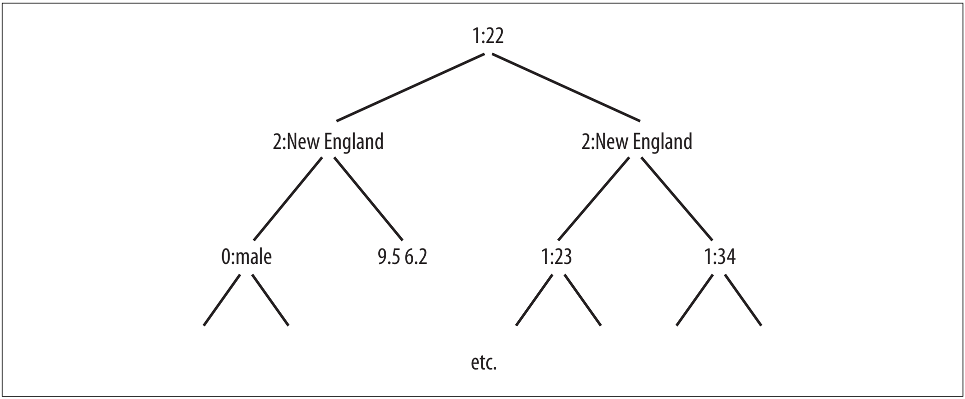


Figure 7-7. Decision tree model of hotness

```
>>> south[10]/sum(south.values( ))
0.055820815183261735
>>> midat[10]/sum(midat.values( ))
0.048972797320600864
```

For this dataset, there are slightly more super-hot people in the South. You can try other things like considering age groups, or testing whether men get better scores than women.

When to Use Decision Trees

Probably the biggest advantage of decision trees is how easy it is to interpret a trained model. After running the algorithm on our example problem, we not only end up with a tree that can make predictions about new users, we also get the list of questions used to make those determinations. From this you can see that, for instance, users who find the site through Slashdot never become paid subscribers, but users who find the site through Google and view at least 20 pages are likely to become premium subscribers. This, in turn, might allow you to alter your advertising strategy to target sites that give you the highest quality traffic. We also learn that certain variables, such as the user's country of origin, are not important in determining the outcome. If data is difficult or expensive to collect and we learn that it is not important, we know that we can stop collecting it.

Unlike some other machine-learning algorithms, decision trees can work with both categorical and numerical data as inputs. In the first example problem, we used a combination of pages viewed with several categorical inputs. Furthermore, while many algorithms require you to prepare or normalize data before you can run them, the code in this chapter will take any list of data containing category or numerical data and build the appropriate decision tree.

Decision trees also allow for probabilistic assignment of data. With some problems, there is not enough information to always make a correct distinction—a decision tree may have a node that has several possibilities and can't be divided any more. The code in this chapter returns a dictionary of the counts of different outcomes, and this information can help us decide how confident we are in the results. Not all algorithms can estimate the probability of an uncertain result.

However, there are definitely drawbacks to the decision tree algorithm used here. While it can be very effective for problems with only a few possible results, it can't be used effectively on datasets with many possibilities. In the first example, the only outcomes are none, basic, and premium. If there were hundreds of outcomes instead, the decision tree would grow very complicated and would probably make poor predictions.

The other big disadvantage of the decision trees described here is that while they can handle simple numerical data, they can only create greater-than/less-than decision points. This makes it difficult for decision trees to classify data where the class is determined by a more complex combination of the variables. For instance, if the results were determined by the differences of two variables, the tree would get very large and would quickly become inaccurate.

In sum, decision trees are probably not a good choice for problems with many numerical inputs and outputs, or with many complex relationships between numerical inputs, such as in interpreting financial data or image analysis. Decision trees are great for datasets with a lot of categorical data and numerical data that has breakpoints. These trees are the best choice if understanding the decision-making process is important; as you've observed, seeing the reasoning can be as important as knowing the final prediction.

Exercises

1. *Result probabilities*. Currently, the `classify` and `mdclassify` functions give their results as total counts. Modify them to give the probabilities of the results being one of the categories.
2. *Missing data ranges*. `mdclassify` allows the use of "None" to specify a missing value. For numerical values the result may not be completely unknown, but may be known to be in a range. Modify `mdclassify` to allow a tuple such as (20,25) in place of a value and traverse down both branches when necessary.
3. *Early stopping*. Rather than pruning the tree, `buildtree` can just stop dividing when it reaches a point where the entropy is not reduced enough. This may not be ideal in some cases, but it does save an extra step. Modify `buildtree` to take a minimum gain parameter and stop dividing the branch if this condition is not met.

4. *Building with missing data*. You built a function that can classify a row with missing data, but what if there is missing data in the training set? Modify `buildtree` so that it will check for missing data and, in cases where it's not possible to send a result down a particular branch, will send it down both branches.
5. *Multiway splits*. (Hard) All the trees built in this chapter are binary decision trees. However, some datasets might create simpler trees if they allowed a node to split into more than two branches. How would you represent this? How would you train the tree?