3 Splitting datasets one feature at a time: decision trees

This chapter covers

- Introducing decision trees
- Measuring consistency in a dataset
- Using recursion to construct a decision tree
- Plotting trees in Matplotlib

Have you ever played a game called Twenty Questions? If not, the game works like this: One person thinks of some object and players try to guess the object. Players are allowed to ask 20 questions and receive only yes or no answers. In this game, the people asking the questions are successively splitting the set of objects they can deduce. A *decision tree* works just like the game Twenty Questions; you give it a bunch of data and it generates answers to the game.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

The decision tree is one of the most commonly used classification techniques; recent surveys claim that it's *the* most commonly used technique.[1] You don't have to know much about machine learning to understand how it works.

If you're not already familiar with decisions trees, the concept is straightforward. Chances are good that you've already seen a decision tree without knowing it. Figure 3.1 shows a flowchart, which is a decision tree. It has *decision blocks* (rectangles) and *terminating blocks* (ovals) where some conclusion has been reached. The right and left arrows coming out of the decision blocks are known as *branches*, and they can lead to other decision blocks or to a terminating block. In this particular example, I made a hypothetical email classification system, which first checks the domain of the sending email address. If this is equal to myEmployer.com, it will classify the email as "Email to read when bored." If it isn't from that domain, it checks to see if the body of the email contains the word *hockey*. If the email contains the word *hockey*, then this email is classified as "Email from friends; read immediately"; if the body doesn't contain the word *hockey*, then it gets classified as "Spam; don't read."

The kNN algorithm in chapter 2 did a great job of classifying, but it didn't lead to any major insights about the data. One of the best things about decision trees is that humans can easily understand the data.

The algorithm you'll build in this chapter will be able to take a set of data, build a decision tree, and draw a tree like the one in figure 3.1. The decision tree does a great job of distilling data into knowledge. With this, you can take a set of unfamiliar data and extract a set of rules. The machine learning will take place as the machine creates these rules from the dataset. Decision trees are often used in expert systems, and the results obtained by using them are often comparable to those from a human expert with decades of experience in a given field.

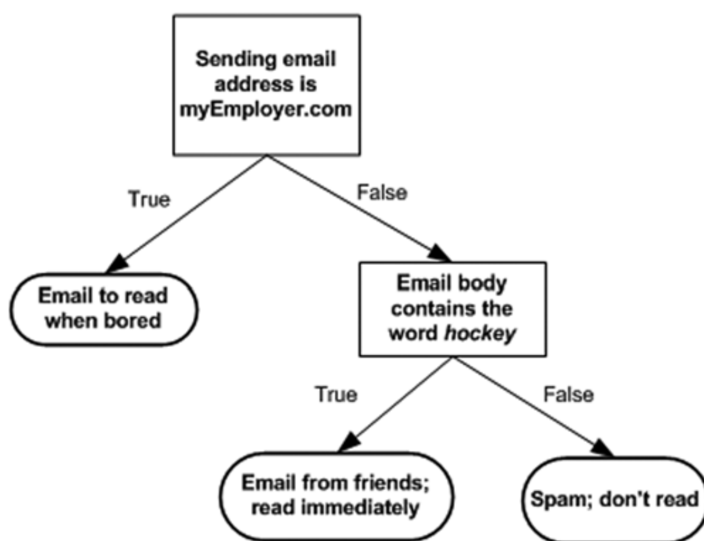


Figure 3.1 A decision tree in flowchart form

[1] Giovanni Seni and John Elder, *Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions,* Synthesis Lectures on Data Mining and Knowledge Discovery (Morgan and Claypool, 2010), 28.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

Now that you know a little of what decision trees are good for, we're going to get into the process of building them from nothing but a pile of data. In the first section, we'll discuss methods used to construct trees and start writing code to construct a tree. Next, we'll address some metrics that we can use to measure the algorithm's success. Finally, we'll use recursion to build our classifier and plot it using Matplotlib. When we have the classifier working, we'll take some data of a contact lens prescription and use our classifier to try to predict what lenses people will need.

3.1 Tree construction

Decision trees

Pros: Computationally cheap to use, easy for humans to understand learned results, missing values OK, can deal with irrelevant features

Cons: Prone to overfitting

Works with: Numeric values, nominal values

In this section we're going to walk through the decision tree–building algorithm, with all its fine details. We'll first discuss the mathematics that decide how to split a dataset using something called *information theory*. We'll then write some code to apply this theory to our dataset, and finally we'll write some code to build a tree.

To build a decision tree, you need to make a first decision on the dataset to dictate which feature is used to split the data. To determine this, you try every feature and measure which split will give you the best results. After that, you'll split the dataset into subsets. The subsets will then traverse down the branches of the first decision node. If the data on the branches is the same class, then you've properly classified it and don't need to continue splitting it. If the data isn't the same, then you need to repeat the splitting process on this subset. The decision on how to split this subset is done the same way as the original dataset, and you repeat this process until you've classified all the data.

Pseudo-code for a function called `createBranch()` would look like this:

```
Check if every item in the dataset is in the same class:
    If so return the class label
    Else
        find the best feature to split the data
        split the dataset
        create a branch node
            for each split
                call createBranch and add the result to the branch node
        return branch node
```

Please note the recursive nature of `createBranch`. It calls itself in the second-to-last line. We'll write this in Python later, but first, we need to address how to split the dataset.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

General approach to decision trees

1. Collect: Any method.
2. Prepare: This tree-building algorithm works only on nominal values, so any continuous values will need to be quantized.
3. Analyze: Any method. You should visually inspect the tree after it is built.
4. Train: Construct a tree data structure.
5. Test: Calculate the error rate with the learned tree.
6. Use: This can be used in any supervised learning task. Often, trees are used to better understand the data.

Some decision trees make a binary split of the data, but we won't do this. If we split on an attribute and it has four possible values, then we'll split the data four ways and create four separate branches. We'll follow the ID3 algorithm, which tells us how to split the data and when to stop splitting it. (See http://en.wikipedia.org/wiki/ID3_algorithm for more information.) We're also going to split on one and only one feature at a time. If our training set has 20 features, how do we choose which one to use first?

See the data in table 3.1. It contains five animals pulled from the sea and asks if they can survive without coming to the surface and if they have flippers. We would like to classify these animals into two classes: fish and not fish. Now we want to decide whether we should split the data based on the first feature or the second feature. To answer this question, we need some quantitative way of determining how to split the data. We'll discuss that next.

	Can survive without coming to surface?	Has flippers?	Fish?
1	Yes	Yes	Yes
2	Yes	Yes	Yes
3	Yes	No	No
4	No	Yes	No
5	No	Yes	No

Table 3.1 Marine animal data

3.1.1 Information gain

We choose to split our dataset in a way that makes our unorganized data more organized. There are multiple ways to do this, and each has its own advantages and disadvantages. One way to organize this messiness is to measure the information. Using

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

information theory, you can measure the information before and after the split. Information theory is a branch of science that's concerned with quantifying information.

The change in information before and after the split is known as the *information gain*. When you know how to calculate the information gain, you can split your data across every feature to see which split gives you the highest information gain. The split with the highest information gain is your best option.

Before you can measure the best split and start splitting our data, you need to know how to calculate the information gain. The measure of information of a set is known as the *Shannon entropy*, or just *entropy* for short. Its name comes from the father of information theory, Claude Shannon.

Claude Shannon

Claude Shannon is considered one of the smartest people of the twentieth century. In William Poundstone's 2005 book *Fortune's Formula*, he wrote this of Claude Shannon:

"There were many at Bell Labs and MIT who compared Shannon's insight to Einstein's. Others found that comparison unfair—unfair to Shannon."[†]

[†] William Poundstone, *Fortune's Formula: The Untold Story of the Scientific Betting System that Beat the Casinos and Wall Street"* (Hill and Wang, 2005), 15.

If the terms *information gain* and *entropy* sound confusing, don't worry. They're meant to be confusing! When Claude Shannon wrote about information theory, John von Neumann told him to use the term *entropy* because people wouldn't know what it meant.

Entropy is defined as the expected value of the information. First, we need to define information. If you're classifying something that can take on multiple values, the information for symbol x^i is defined as

$$l(x_i) = log_2 p(x_i)$$

where $p(x^i)$ is the probability of choosing this class.

To calculate entropy, you need the expected value of all the information of all possible values of our class. This is given by

$$H = -\sum_{i=1}^{n} p(x_i) log_2 p(x_i)$$

where n is the number of classes.

Let's see how to calculate this in Python. To start, you'll create a file called trees.py. Insert the code from the following listing into trees.py. This listing will do entropy calculations on a given dataset for you.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

Listing 3.1 Function to calculate the Shannon entropy of a dataset

```
from math import log

def calcShannonEnt(dataSet):
    numEntries = len(dataSet)
    labelCounts = {}
    for featVec in dataSet:
        currentLabel = featVec[-1]
        if currentLabel not in labelCounts.keys():
        labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1
    shannonEnt = 0.0
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries
        shannonEnt -= prob * log(prob,2)
    return shannonEnt
```

1 Create dictionary of all possible classes

2 Logarithm base 2

The code in listing 3.1 is straightforward. First, you calculate a count of the number of instances in the dataset. This could have been calculated inline, but it's used multiple times in the code, so an explicit variable is created for it. Next, you create a dictionary whose keys are the values in the final column. **1** If a key was not encountered previously, one is created. For each key, you keep track of how many times this label occurs. Finally, you use the frequency of all the different labels to calculate the probability of that label. This probability is used to calculate the Shannon entropy, **2** and you sum this up for all the labels. Let's try out this entropy stuff.

The simple data about fish identification from table 3.1 is provided in the trees.py file by utilizing the `createDataSet()` function. You can enter it yourself:

```
def createDataSet():
       dataSet = [[1, 1, 'yes'],
               [1, 1, 'yes'],
               [1, 0, 'no'],
               [0, 1, 'no'],
               [0, 1, 'no']]
labels = ['no surfacing','flippers']
return dataSet, labels
```

Enter the following in your Python shell:

```
>>> reload(trees.py)
>>> myDat,labels=trees.createDataSet()
>>> myDat
[[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]
>>> trees.calcShannonEnt(myDat)
0.97095059445466858
```

The higher the entropy, the more mixed up the data is. Let's make the data a little messier and see how the entropy changes. We'll add a third class, which is called `maybe`, and see how the entropy changes:

```
>>> myDat[0][-1]='maybe'
>>> myDat
[[1, 1, 'maybe'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]
>>> trees.calcShannonEnt(myDat)
1.3709505944546687
```

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

Let's split the dataset in a way that will give us the largest information gain. We won't know how to do that unless we actually split the dataset and measure the information gain.

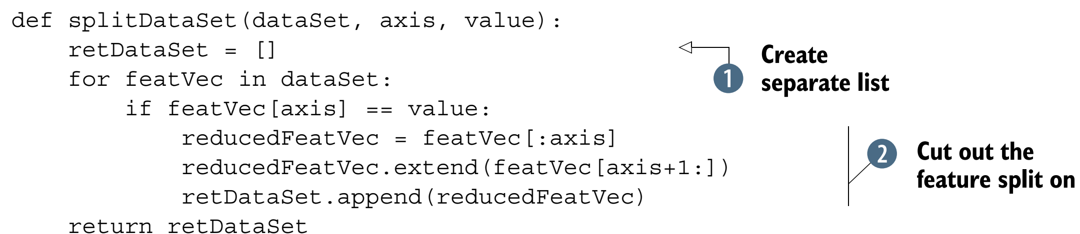
Another common measure of disorder in a set is the Gini impurity,[2] which is the probability of choosing an item from the set and the probability of that item being misclassified. We won't get into the Gini impurity. Instead, we'll move on to splitting the dataset and building the tree.

3.1.2 Splitting the dataset

You just saw how to measure the amount of disorder in a dataset. For our classifier algorithm to work, you need to measure the entropy, split the dataset, measure the entropy on the split sets, and see if splitting it was the right thing to do. You'll do this for all of our features to determine the best feature to split on. Think of it as a two-dimensional plot of some data. You want to draw a line to separate one class from another. Should you do this on the X-axis or the Y-axis? The answer is what you're trying to find out here.

To see this in action, open your editor and add the following code to trees.py.

Listing 3.2 Dataset splitting on a given feature




```
def splitDataSet(dataSet, axis, value):
    retDataSet = []
    for featVec in dataSet:
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
    return retDataSet
```

The code in listing 3.2 takes three inputs: the dataset we'll split, the feature we'll split on, and the value of the feature to return. Most of the time in Python, you don't have to worry about memory or allocation. Python passes lists by reference, so if you modify a list in a function, the list will be modified everywhere. To account for this, you create a new list at the beginning. ❶ You create a new list each time because you'll be calling this function multiple times on the same dataset and you don't want the original dataset modified. Our dataset is a list of lists; you iterate over every item in the list and if it contains the value you're looking for, you'll add it to your newly created list. Inside the `if` statement, you cut out the feature that you split on. ❷ This will be more obvious in the next section, but think of it this way: once you've split on a feature, you're finished with that feature. You used the `extend()` and `append()` methods of the Python list type. There's an important difference between these two methods when dealing with multiple lists.

Assume you have two lists, a and b:

---

[2] For more information, you should check out *Introduction to Data Mining* by Pan-Ning Tan, Vipin Kumar, and Michael Steinbach; Pearson Education (Addison-Wesley, 2005), 158.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

```
>>> a=[1,2,3]
>>> b=[4,5,6]
>>> a.append(b)
>>> a
[1, 2, 3, [4, 5, 6]]
```

If you do `a.append(b)`, you have a list with four elements, and the fourth element is a list. However, if you do

```
>>> a=[1,2,3]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6]
```

you now have one list with all the elements from a and b.

Let's try out the `splitDataSet()` function on our simple example. Add the code from listing 3.2 to trees.py, and type in the following at your Python shell:

```
>>> reload(trees)
<module 'trees' from 'trees.pyc'>
>>> myDat,labels=trees.createDataSet()
>>> myDat
[[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]
>>> trees.splitDataSet(myDat,0,1)
[[1, 'yes'], [1, 'yes'], [0, 'no']]
>>> trees.splitDataSet(myDat,0,0)
[[1, 'no'], [1, 'no']]
```

You're now going to combine the Shannon entropy calculation and the `splitDataSet()` function to cycle through the dataset and decide which feature is the best to split on. Using the entropy calculation tells you which split best organizes your data.

Open your text editor and add the code from the following listing to trees.py.

Listing 3.3 Choosing the best feature to split on

```
def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1
    baseEntropy = calcShannonEnt(dataSet)
    bestInfoGain = 0.0; bestFeature = -1
    for i in range(numFeatures):
        featList = [example[i] for example in dataSet]
        uniqueVals = set(featList)
        newEntropy = 0.0
        for value in uniqueVals:
            subDataSet = splitDataSet(dataSet, i, value)
            prob = len(subDataSet)/float(len(dataSet))
            newEntropy += prob * calcShannonEnt(subDataSet)
        infoGain = baseEntropy - newEntropy
        if (infoGain > bestInfoGain):
            bestInfoGain = infoGain
            bestFeature = I
    return bestFeature
```

1 Create unique list of class labels

2 Calculate entropy for each split

3 Find the best information gain

The code in listing 3.3 is the function `chooseBestFeatureToSplit()`. As you can guess, it chooses the feature that, when split on, best organizes your data. The functions from

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

listing 3.2 and listing 3.1 are used in this function. We've made a few assumptions about the data. The first assumption is that it comes in the form of a list of lists, and all these lists are of equal size. The next assumption is that the last column in the data or the last item in each instance is the class label of that instance. You use these assumptions in the first line of the function to find out how many features you have available in the given dataset. We didn't make any assumption on the type of data in the lists. It could be a number or a string; it doesn't matter.

The next part of the code in listing 3.3 calculates the Shannon entropy of the whole dataset before any splitting has occurred. This gives you the base disorder, which you'll later compare to the post split disorder measurements. The first `for` loop loops over all the features in our dataset. You use list comprehensions to create a list of all the ith entries in our dataset, or all the possible values present in the data. ❶ Next, you use the Python native set data type. Sets are like lists, but a value can occur only once. Creating a new set from a list is one of the fastest ways of getting the unique values out of list in Python.

Next, you go through all the unique values of this feature and split the data for each feature. ❷ The new entropy is calculated and summed up for all the unique values of that feature. The information gain is the reduction in entropy or the reduction in messiness. I hope entropy makes sense when put in terms of reduction of disorder. Finally, you compare the information gain among all the features and return the index of the best feature to split on. ❸

Now let's see this in action. After you enter the code from listing 3.3 into trees.py, type the following at your Python shell:

```
>>> reload(trees)
<module 'trees' from 'trees.py'>
>>> myDat,labels=trees.createDataSet()
>>> trees.chooseBestFeatureToSplit(myDat)
0
>>> myDat
[[1, 1, 'yes'], [1, 1, 'yes'], [1, 0, 'no'], [0, 1, 'no'], [0, 1, 'no']]
```

What just happened? The code told you that the 0th feature was the best feature to split on. Is that right? Does that make any sense? It's the same data from table 3.1, so let's look at table 3.1, or the data from the variable `myDat`. If you split on the first feature, that is, put everything where the first feature is 1 in one group and everything where the first feature is 0 in another group, how consistent is the data? If you do that, the group where the first feature is 1 will have two yeses and one no. The other group will have zero yeses and two nos. What if you split on the second feature? The first group will have two yeses and two nos. The second group will have zero yeses and one no. The first split does a better job of organizing the data. If you're not convinced, you can use the `calcShannonEntropy()` function from listing 3.1 to test it.

Now that you can measure how organized a dataset is and you can split the data, it's time to put all of this together and build the decision tree.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

3.1.3 Recursively building the tree

You now have all the components you need to create an algorithm that makes decision trees from a dataset. It works like this: you start with our dataset and split it based on the best attribute to split. These aren't binary trees, so you can handle more than two-way splits. Once split, the data will traverse down the branches of the tree to another node. This node will then split the data again. You're going to use the principle of recursion to handle this.

You'll stop under the following conditions: you run out of attributes on which to split or all the instances in a branch are the same class. If all instances have the same class, then you'll create a leaf node, or terminating block. Any data that reaches this leaf node is deemed to belong to the class of that leaf node. This process can be seen in figure 3.2.

The first stopping condition makes this algorithm tractable, and you can even set a bound on the maximum number of splits you can have. You'll encounter other decision-tree algorithms later, such as C4.5 and CART. These do not "consume" the features at each split. This creates a problem for these algorithms because they split

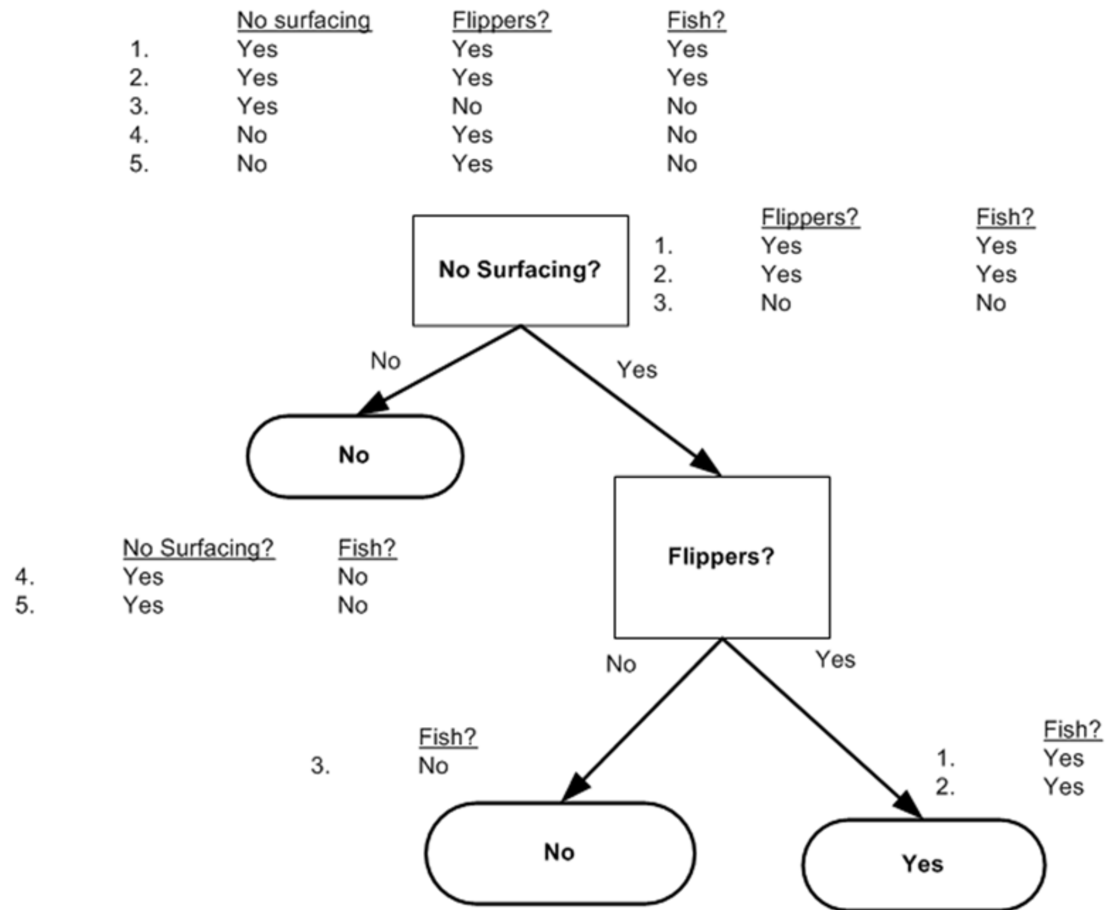


Figure 3.2 **Data paths while splitting**

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

the data, but the number of features doesn't decrease at each split. Don't worry about that for now. You can simply count the number of columns in our dataset to see if you've run out of attributes. If our dataset has run out of attributes but the class labels are not all the same, you must decide what to call that leaf node. In this situation, you'll take a majority vote.

Open your editor of choice. Before you add the next function, you need to add the following line to the top of trees.py: `import operator`. Now, add the following function to trees.py:

```
def majorityCnt(classList):
    classCount={}
    for vote in classList:
        if vote not in classCount.keys(): classCount[vote] = 0
        classCount[vote] += 1
    sortedClassCount = sorted(classCount.iteritems(),
     key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]
```

This function may look familiar; it's similar to the voting portion of `classify0` from chapter 2. This function takes a list of class names and then creates a dictionary whose keys are the unique values in `classList`, and the object of the dictionary is the frequency of occurrence of each class label from `classList`. Finally, you use the operator to sort the dictionary by the keys and return the class that occurs with the greatest frequency.

Open trees.py in your editor and add the code from the following listing.

Listing 3.4 Tree-building code

```
def createTree(dataSet,labels):
    classList = [example[-1] for example in dataSet]
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    if len(dataSet[0]) == 1:
        return majorityCnt(classList)
    bestFeat = chooseBestFeatureToSplit(dataSet)
    bestFeatLabel = labels[bestFeat]
    myTree = {bestFeatLabel:{}}
    del(labels[bestFeat])
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]
        myTree[bestFeatLabel][value] = createTree(splitDataSet\
                         (dataSet, bestFeat, value),subLabels)
    return myTree
```

1. **Stop when all classes are equal**
2. **When no more features, return majority**
3. **Get list of unique values**

The code in listing 3.4 takes two inputs: the dataset and a list of labels. The list of labels contains a label for each of the features in the dataset. The algorithm could function without this, but it would be difficult to make any sense of the data. All of the previous assumptions about the dataset still hold. You first create a list of all the class labels in our dataset and call this classList. The first stopping condition is that if all the

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

class labels are the same, then you return this label. ❶ The second stopping condition is the case when there are no more features to split. ❷ If you don't meet the stopping conditions, then you use the function created in listing 3.3 to choose the best feature. Next, you create your tree.

You'll use the Python dictionary to store the tree. You could have created a special data type, but it's not necessary. The myTree dictionary will be used to store the tree, and you'll see how that works soon. You get all the unique values from the dataset for our chosen feature: bestFeat. ❸ The unique value code uses sets and is similar to a few lines in listing 3.3.

Finally, you iterate over all the unique values from our chosen feature and recursively call `createTree()` for each split of the dataset. This value is inserted into our myTree dictionary, so you end up with a lot of nested dictionaries representing our tree. Before we get into the nesting, note that the `subLabels = labels[:]` line makes a copy of labels and places it in a new list called subLabels. You do this because Python passes lists by reference and you'd like the original list to be the same every time you call `createTree()`.

Let's try out this code. After you add the code from listing 3.4 to trees.py, enter the following in your Python shell:

```
>>> reload(trees)
<module 'trees' from 'trees.pyc'>
>>> myDat,labels=trees.createDataSet()
>>> myTree = trees.createTree(myDat,labels)
>>> myTree
{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}}
```

The variable `myTree` contains the nested dictionaries, which you're using to represent our tree structure. Reading left to right, the first key, `'no surfacing'`, is the name of the first feature that was split by the create tree. The value of this key is another dictionary. This second dictionary's keys are the splits of the `'no surfacing'` feature. The values of these keys are the children of the `'no surfacing'` node. The values are either a class label or another dictionary. If the value is a class label, then that child is a leaf node. If the value is another dictionary, then that child node is a decision node and the format repeats itself. In our example, we have three leaf nodes and two decision nodes.

Now that you've properly constructed the tree, you need to display it so that humans can properly understand the information.

3.2 Plotting trees in Python with Matplotlib annotations

The tree you made in the previous section is great, but it's a little difficult to visualize. In this section, we'll use Matplotlib to create a tree you can look at. One of the greatest strengths of decision trees is that humans can easily understand them. The plotting library we used in the previous chapter is extremely powerful. Unfortunately, Python doesn't include a good tool for plotting trees, so we'll make our own. We'll write a program to draw a decision tree like the one in figure 3.3.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

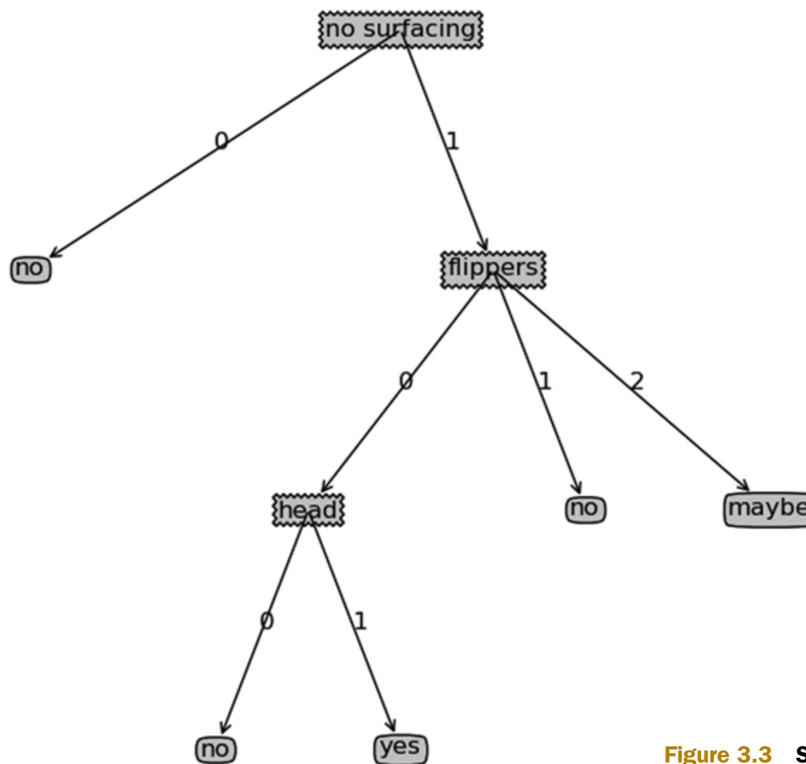


Figure 3.3 Sample decision tree

3.2.1 Matplotlib annotations

Matplotlib has a great tool, called *annotations*, that can add text near data in a plot. Annotations are usually used to explain some part of the data. But having the text on top of the data looks ugly, so the tool has a built-in arrow that allows you to draw the text a safe distance away from the data yet show what data you're talking about. Figure 3.4 shows this in action. We have a point at (0.2, 0.1), and we placed some text at (0.35, 0.3) and an arrow pointing to the point at (0.2, 0.1).

> **Plot or graph?**
>
> Why use the word *plot*? Why not use the word *graph* for talking about showing data in an image? In some disciplines, the word *graph* has a different meaning. In applied mathematics, it's a representation of a set of objects (vertices) connected by edges. Any combination of the vertices can be connected by edges. In computer science, a graph is a data structure that's used to represent the concept from mathematics.

We're going to hijack the annotations and use them for our tree plotting. You can color in the box of the text and give it a shape you like. Next, you can flip the arrow and have it point from the data point to the text box. Open your text editor and create a new file called treePlotter.py. Add the code from the following listing.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

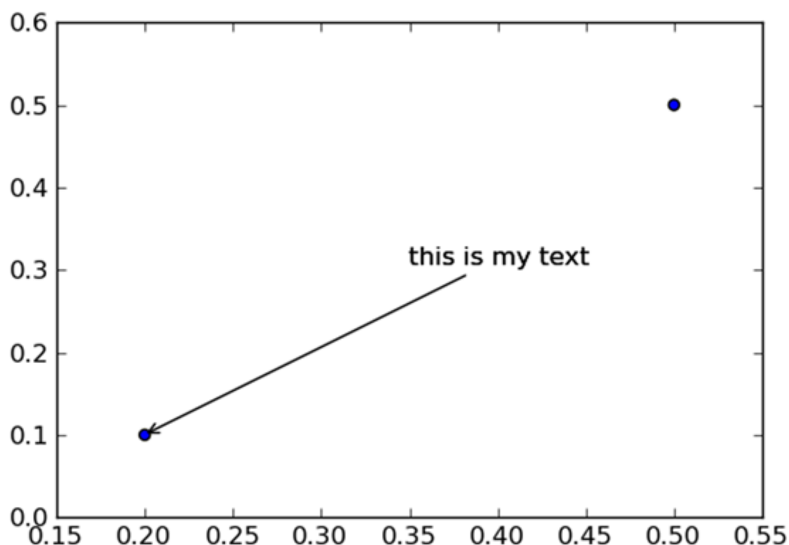


Figure 3.4 Matplotlib annotations demonstration

Listing 3.5 Plotting tree nodes with text annotations

```
import matplotlib.pyplot as plt

decisionNode = dict(boxstyle="sawtooth", fc="0.8")
leafNode = dict(boxstyle="round4", fc="0.8")
arrow_args = dict(arrowstyle="<-")

def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    createPlot.ax1.annotate(nodeTxt, xy=parentPt,
xycoords='axes fraction',
    xytext=centerPt, textcoords='axes fraction',
    va="center", ha="center", bbox=nodeType, arrowprops=arrow_args)

def createPlot():
    fig = plt.figure(1, facecolor='white')
    fig.clf()
    createPlot.ax1 = plt.subplot(111, frameon=False)
    plotNode('a decision node', (0.5, 0.1), (0.1, 0.5), decisionNode)
    plotNode('a leaf node', (0.8, 0.1), (0.3, 0.8), leafNode)
    plt.show()
```

(1) Define box and arrow formatting

(2) Draws annotations with arrows

If `createPlot()` doesn't look like `createPlot()` in the example text file, don't worry. You'll change it later. The code in the listing begins by defining some constants that you'll use for formatting the nodes. (1) Next, you create the `plotNode()` function, which actually does the drawing. It needs a plot to draw these on, and the plot is the global variable `createPlot.ax1`. In Python, all variables are global by default, and if you know what you're doing, this won't get you into trouble. Lastly, you have the `createPlot()` function, which is the master. Here, you create a new figure, clear it, and then draw on two nodes to demonstrate the different types of nodes you'll use in plotting your tree.

To give this code a try, open your Python shell and import the treePlotter file.

```
>>> import treePlotter
>>> treePlotter.createPlot()
```

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

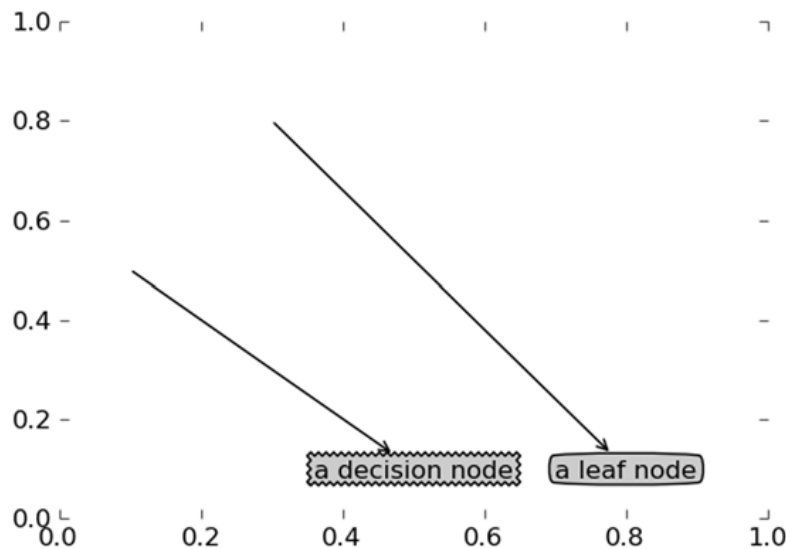


Figure 3.5 Example of the `plotNode` function

You should see something that looks like figure 3.5. You can alter the points in `plotNode()` ❷ to see how the X,Y position changes.

Now that you can plot the nodes, you're ready to combine more of these to plot a whole tree.

3.2.2 *Constructing a tree of annotations*

You need a strategy for plotting this tree. You have X and Y coordinates. Now, where do you place all the nodes? You need to know how many leaf nodes you have so that you can properly size things in the X direction, and you need to know how many levels you have so you can properly size the Y direction. You're going to create two new functions to get the two items you're looking for. The next listing has the functions `getNumLeafs()` and `getTreeDepth()`. Add these two functions to treePlotter.py.

Listing 3.6 Identifying the number of leaves in a tree and the depth

```
def getNumLeafs(myTree):
    numLeafs = 0
    firstStr = myTree.keys()[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':
            numLeafs += getNumLeafs(secondDict[key])
        else:   numLeafs +=1
    return numLeafs

def getTreeDepth(myTree):
    maxDepth = 0
    firstStr = myTree.keys()[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':
```

❶ **Test if node is dictionary**

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

```
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else:   thisDepth = 1
        if thisDepth > maxDepth: maxDepth = thisDepth
    return maxDepth
```

The two functions in listing 3.6 have the same structure, which you'll use again later. The structure is built around how you store the tree in a Python dictionary. The first key is the label of the first split, and the values associated with that key are the children of the first node. You get out the first key and value, and then you iterate over all of the child nodes. You test to see if the child nodes are dictionaries by using the Python `type()` method. ❶ If the child node is of type dict, then it is another decision node and you must recursively call your function. The `getNumLeafs()` function traverses the entire tree and counts only the leaf nodes; then it returns this number. The second function, `getTreeDepth()`, counts the number of times you hit a decision node. The stopping condition is a leaf node, and once this is reached you back out of your recursive calls and increment the count. To save you some time, I added a simple function to output premade trees. This will save you the trouble of making a tree from data every time during testing.

Enter the following into treePlotter.py:

```
def retrieveTree(i):
    listOfTrees =[{'no surfacing': {0: 'no', 1: {'flippers': \
                    {0: 'no', 1: 'yes'}}}},
                  {'no surfacing': {0: 'no', 1: {'flippers': \
                    {0: {'head': {0: 'no', 1: 'yes'}}, 1: 'no'}}}}
                  ]
    return listOfTrees[i]
```

Save treePlotter.py and enter the following into your Python shell:

```
>>> reload(treePlotter)
<module 'treePlotter' from 'treePlotter.py'>
>>> treePlotter.retrieveTree (1)
{'no surfacing': {0: 'no', 1: {'surfacing': {0: {'head': {0: 'no', 1:
     'yes'}}, 1: 'no'}}}}
>>> myTree = treePlotter.retrieveTree (0)
>>> treePlotter.getNumLeafs(myTree)
3
>>> treePlotter.getTreeDepth(myTree)
2
```

The `retrieveTree()` function pulls out a predefined tree for testing. You can see that `getNumLeafs()` returns three leaves, which is what tree 0 has. The function `getTreeDepth()` also returns the proper number levels.

Now you can put all of these elements together and plot the whole tree. When you're finished, the tree will look something like the one in figure 3.6 but without the labels on the X and Y axes.

Open your text editor and enter the code from the following listing into treePlotter.py. Note that you probably already have a version of `treePlotter()`. Please change it to look like the following code.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

Listing 3.7 The `plotTree` function

```
def plotMidText(cntrPt, parentPt, txtString):
    xMid = (parentPt[0]-cntrPt[0])/2.0 + cntrPt[0]
    yMid = (parentPt[1]-cntrPt[1])/2.0 + cntrPt[1]
    createPlot.ax1.text(xMid, yMid, txtString)

def plotTree(myTree, parentPt, nodeTxt):
    numLeafs = getNumLeafs(myTree)
    getTreeDepth(myTree)
    firstStr = myTree.keys()[0]
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW,\
                               plotTree.yOff)
    plotMidText(cntrPt, parentPt, nodeTxt)
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
    secondDict = myTree[firstStr]
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':
            plotTree(secondDict[key],cntrPt,str(key))
        else:
            plotTree.xOff = plotTree.xOff + 1.0/plotTree.totalW
            plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff),
                cntrPt, leafNode)
            plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))
    plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD

def createPlot(inTree):
    fig = plt.figure(1, facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
    createPlot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plotTree.totalW = float(getNumLeafs(inTree))
    plotTree.totalD = float(getTreeDepth(inTree))
    plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0;
    plotTree(inTree, (0.5,1.0), '')
    plt.show()
```

1. Plots text between child and parent
2. Get the width and height
3. Plot child value
4. Decrement Y offset

The `createPlot()` function is the main function you'll use, and it calls `plotTree()`, which in turns calls many of the previous functions and `plotMidText()`. The function `plotTree()` does the majority of the work. The first thing that happens in `plotTree()` is the calculation of width and height of the tree. (2) Two global variables are set up to store the width (`plotTree.totalW`) and depth of the tree (`plotTree.totalD`). These variables are used in centering the tree nodes vertically and horizontally. The `plotTree()` function gets called recursively like `getNumLeafs()` and `getTreeDepth()` from listing 3.6. The width of the tree is used to calculate where to place the decision node. The idea is to place this in the middle of all the leaf nodes below it, not place it in the middle of its children. Also note that you use two global variables to keep track of what has already been plotted and the appropriate coordinate to place the next node. These values are stored in `plotTree.xOff` and `plotTree.yOff`. Another thing to point out is that you're plotting everything on the x-axis from 0.0 to 1.0 and on the y-axis from 0.0 to 1.0. Figure 3.6 has these values labeled for your convenience. The

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

center point for the current node is plotted with its total width split by the total number of leafs in the global tree. This allows you to split the x-axis into as many segments as you have leaves. The beautiful thing about plotting everything in terms of the image width is that you can resize the image, and the node will be redrawn in its proper place. If this was drawn in terms of pixels, that wouldn't be the case. You couldn't resize the image as easily.

Next, you plot the child value or the value for the feature for the split going down that branch. ❸ The code in `plotMidText()` calculates the midpoint between the parent and child nodes and puts a simple text label in the middle. ❶

Next, you decrement the global variable `plotTree.yOff` to make a note that you're about to draw children nodes. ❹ These nodes could be leaf nodes or other decision nodes, but you need to keep track of this. You decrement rather than increment because you start drawing from the top of the image and draw downward. You next recursively go through the tree in a similar fashion as the `getNumLeafs()` and `getTreeDepth()` functions. If a node is a leaf node, you draw a leaf node. If not, you recursively call `plotTree()` again. Finally, after you finish plotting the child nodes, you increment the global Y offset.

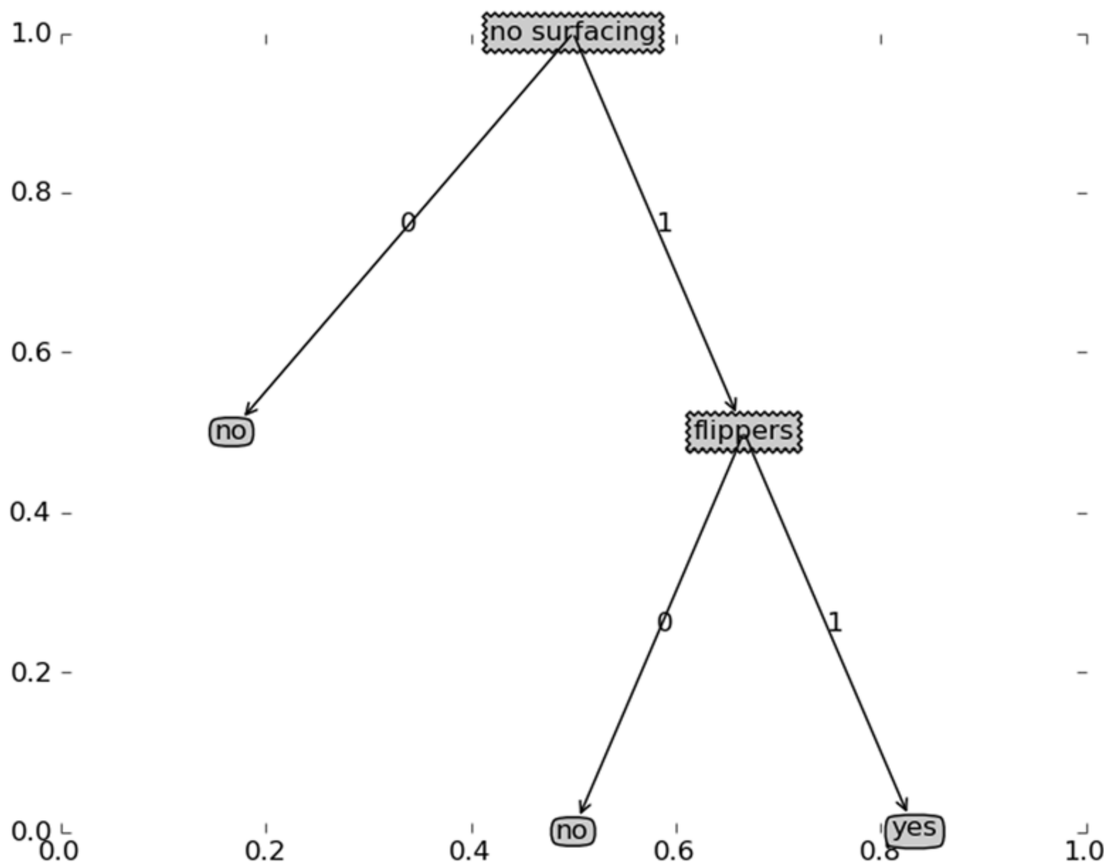


Figure 3.6 Tree plotting of simple dataset showing figure position axes

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

The last function in listing 3.7 is `createPlot()`, which handles setting up the image, calculating the global tree size, and kicking off the recursive `plotTree()` function.

Let's see this in action. After you add the function to treePlotter.py, type the following in your Python shell:

```
>>> reload(treePlotter)
<module 'treePlotter' from 'treePlotter.pyc'>
>>> myTree=treePlotter.retrieveTree (0)
>>> treePlotter.createPlot(myTree)
```

You should see something like figure 3.6 without the axis labels. Now let's alter the dictionary and plot it again.

```
>>> myTree['no surfacing'][3]='maybe'
>>> myTree
{'no surfacing ': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}, 3:
     'maybe'}}
>>> treePlotter.createPlot(myTree)
```

You should see something that looks like figure 3.7 (and a lot like a headless stick figure.) Feel free to play around with the tree data structures and plot them out.

Now that you can build a decision tree and plot out the tree, you can to put it to use and see what you can learn from some data and this algorithm.

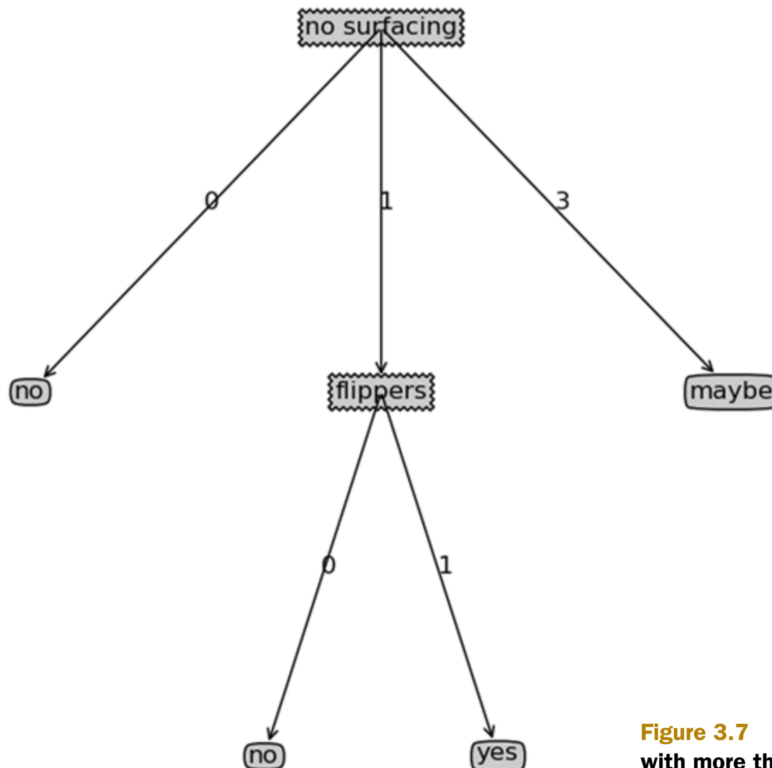


Figure 3.7 Tree plotting with more than two splits

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

3.3 Testing and storing the classifier

The main focus of the first section of this book is on classification. We've done a lot of work in this chapter so far building the tree from data and plotting the tree so a human can make some sense of the data, but we haven't yet done any classification.

In this section, you'll build a classifier that uses our tree, and then you'll see how to persist that classifier on disk for longer storage in a real application. Finally, you'll put our decision tree code to use on some real data to see if you can predict what type of contact lenses a person should use.

3.3.1 Test: using the tree for classification

You want to put our tree to use doing some classification after you've learned the tree from our training data, but how do you do that? You need our tree and the label vector that you used in creating the tree. The code will then take the data under test and compare it against the values in the decision tree. It will do this recursively until it hits a leaf node; then it will stop because it has arrived at a conclusion.

To see this in action, open your text editor and add the code in the following listing to trees.py.

Listing 3.8 Classification function for an existing decision tree

```
def classify(inputTree,featLabels,testVec):
    firstStr = inputTree.keys()[0]
    secondDict = inputTree[firstStr]
    featIndex = featLabels.index(firstStr)
    for key in secondDict.keys():
        if testVec[featIndex] == key:
            if type(secondDict[key]).__name__=='dict':
                classLabel = classify(secondDict[key],featLabels,testVec)
            else:   classLabel = secondDict[key]
    return classLabel
```

❶ **Translate label string to index**

The code in listing 3.8 follows the same format as the other recursive functions in this chapter. A problem with storing your data with the label as the feature's identifier is that you don't know where this feature is in the dataset. To clear this up, you first split on the "no surfacing" attribute, but where is that in the dataset? Is it first or second? The Labels list will tell you this. You use the `index` method to find out the first item in this list that matches `firstStr`. ❶ With that in mind, you can recursively travel the tree, comparing the values in `testVec` to the values in the tree. If you reach a leaf node, you've made your classification and it's time to exit.

After you've added the code in listing 3.8 to your trees.py file, enter the following in your Python shell:

```
>>> myDat,labels=trees.createDataSet()
>>> labels
['no surfacing', 'flippers']
>>> myTree=treePlotter.retrieveTree (0)
>>> myTree
{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}}
```

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

```
>>> trees.classify(myTree,labels,[1,0])
'no'
>>> trees.classify(myTree,labels,[1,1])
'yes'
```

Compare these results to figure 3.6. You have a first node called "no surfacing" that has two children, one called 0, which has a label of "no", and one that's another decision node called "flippers". This checks out. The "flippers" node had two children. Is this the same as between the tree you plotted and the tree data structure? Yes.

Now that you've built a classifier, it would be nice to be able to store this so you don't have to rebuild the tree every time you want to do classification.

3.3.2 Use: persisting the decision tree

Building the tree is the majority of the work. It may take a few seconds with our small datasets, but, with large datasets, this can take a long time. When it's time to classify items with a tree, you can do it quickly. It would be a waste of time to build the tree every time you wanted to make a classification. To get around this, you're going to use a Python module, which is properly named *pickle*, to serialize objects, as shown in the following listing. Serializing objects allows you to store them for later use. Serializing can be done with any object, and dictionaries work as well.

Listing 3.9 Methods for persisting the decision tree with pickle

```
def storeTree(in
putTree,filename):
    import pickle
    fw = open(filename,'w')
    pickle.dump(inputTree,fw)
    fw.close()

def grabTree(filename):
    import pickle
    fr = open(filename)
    return pickle.load(fr)
```

You can experiment with this in your Python shell by typing in the following:

```
>>> trees.storeTree(myTree,'classifierStorage.txt')
>>> trees.grabTree('classifierStorage.txt')
{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}}
```

Now you have a way of persisting your classifier so that you don't have to relearn it every time you want to classify something. This is another advantage of decision trees over another machine learning algorithm like kNN from chapter 2; you can distill the dataset into some knowledge, and you use that knowledge only when you want to classify something. Let's use the tools you've learned thus far on the Lenses dataset.

3.4 Example: using decision trees to predict contact lens type

In this section, we'll go through an example that predicts the contacts lens type that should be prescribed. You'll take a small dataset and see if you can learn anything

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

from it. You'll see if a decision tree can give you any insight as to how the eye doctor prescribes contact lenses. You can predict the type of lenses people will use and understand the underlying processes with a decision tree.

Example: using decision trees to predict contact lens type

1. Collect: Text file provided.
2. Prepare: Parse tab-delimited lines.
3. Analyze: Quickly review data visually to make sure it was parsed properly. The final tree will be plotted with `createPlot()`.
4. Train: Use `createTree()` from section 3.1.
5. Test: Write a function to descend the tree for a given instance.
6. Use: Persist the tree data structure so it can be recalled without building the tree; then use it in any application.

The Lenses dataset[3] is one of the more famous datasets. It's a number of observations based on patients' eye conditions and the type of contact lenses the doctor prescribed. The classes are hard, soft, and no contact lenses. The data is from the UCI database repository and is modified slightly so that it can be displayed easier. The data is stored in a text file with the source code download.

You can load the data by typing the following into your Python shell:

```
>>> fr=open('lenses.txt')
>>> lenses=[inst.strip().split('\t') for inst in fr.readlines()]
>>> lensesLabels=['age', 'prescript', 'astigmatic', 'tearRate']
>>> lensesTree = trees.createTree(lenses,lensesLabels)
>>> lensesTree
{'tearRate': {'reduced': 'no lenses', 'normal': {'astigmatic': {'yes':
{'prescript': {'hyper': {'age': {'pre': 'no lenses', 'presbyopic':
'no lenses', 'young':'hard'}}, 'myope': 'hard'}}, 'no': {'age': {'pre':
'soft', 'presbyopic': {'prescript': {'hyper': 'soft', 'myope':
'no lenses'}}, 'young': 'soft'}}}}}
>>> treePlotter.createPlot(lensesTree)
```

That tree looks difficult to read as a line of text; it's a good thing you have a way to plot it. The tree plotted using our `createPlot()` function is shown in figure 3.8. If you follow the different branches of the tree, you can see what contact lenses should be prescribed to a given individual. One other conclusion you can draw from figure 3.8 is that a doctor has to ask at most four questions to determine what type of lenses a patient will need.

[3] The dataset is a modified version of the Lenses dataset retrieved from the UCI Machine Learning Repository November 3, 2010 [http://archive.ics.uci.edu/ml/machine-learning-databases/lenses/]. The source of the data is Jadzia Cendrowska and was originally published in "PRISM: An algorithm for inducing modular rules," in *International Journal of Man-Machine Studies* (1987), 27, 349–70.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

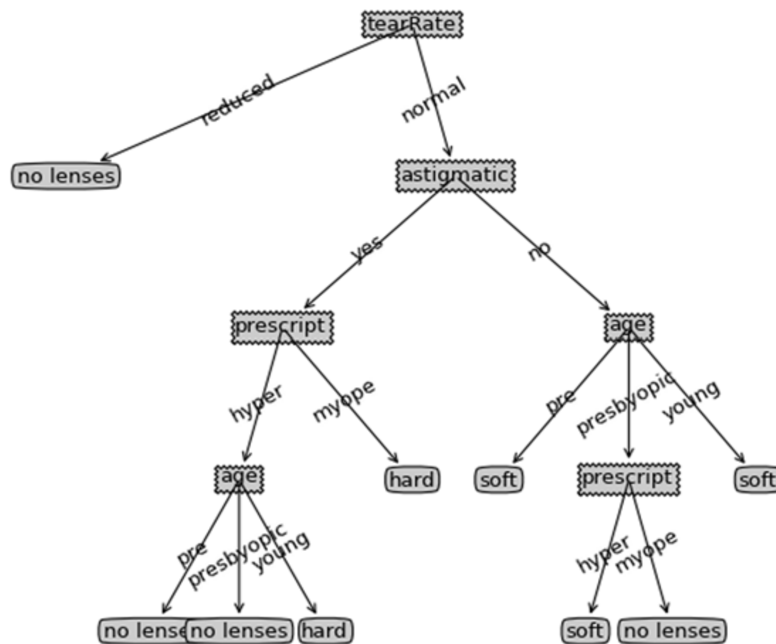


Figure 3.8 Decision tree generated by the ID3 algorithm

The tree in figure 3.8 matches our data well; however, it probably matches our data too well. This problem is known as *overfitting*. In order to reduce the problem of overfitting, we can prune the tree. This will go through and remove some leaves. If a leaf node adds only a little information, it will be cut off and merged with another leaf. We'll investigate this further when we revisit decision trees in chapter 9.

In chapter 9 we'll also investigate another decision tree algorithm called CART. The algorithm we used in this chapter, ID3, is good but not the best. ID3 can't handle numeric values. We could use continuous values by quantizing them into discrete bins, but ID3 suffers from other problems if we have too many splits.

3.5 Summary

A decision tree classifier is just like a work-flow diagram with the terminating blocks representing classification decisions. Starting with a dataset, you can measure the inconsistency of a set or the entropy to find a way to split the set until all the data belongs to the same class. The ID3 algorithm can split nominal-valued datasets. Recursion is used in tree-building algorithms to turn a dataset into a decision tree. The tree is easily represented in a Python dictionary rather than a special data structure.

Cleverly applying Matplotlib's annotations, you can turn our tree data into an easily understood chart. The Python Pickle module can be used for persisting our tree. The contact lens data showed that decision trees can try too hard and overfit a dataset. This overfitting can be removed by pruning the decision tree, combining adjacent leaf nodes that don't provide a large amount of information gain.

There are other decision tree–generating algorithms. The most popular are C4.5 and CART. CART will be addressed in chapter 9 when we use it for regression.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

The first two chapters in this book have drawn hard conclusions about data such as "This data instance is in this class!" What if we take a softer approach, such as "Well, I'm not quite sure where that data should go. Maybe here? Maybe there?" What if we assign a probability to a data instance belonging to a given class? This will be the focus of the next chapter.

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>

4 Classifying with probability theory: naïve Bayes

This chapter covers

- Using probability distributions for classification
- Learning the naïve Bayes classifier
- Parsing data from RSS feeds
- Using naïve Bayes to reveal regional attitudes

In the first two chapters we asked our classifier to make hard decisions. We asked for a definite answer for the question "Which class does this data instance belong to?" Sometimes the classifier got the answer wrong. We could instead ask the classifier to give us a best guess about the class and assign a probability estimate to that best guess.

Probability theory forms the basis for many machine-learning algorithms, so it's important that you get a good grasp on this topic. We touched on probability a bit in chapter 3 when we were calculating the probability of a feature taking a given

Licensed to Brahim Chaibdraa <chaib@iad.ift.ulaval.ca>