Machine Learning in Python®: Essential Techniques for Predictive Analysis
By Michael Bowles

CHAPTER

6

Ensemble Methods

Ensemble methods stem from the observation that multiple models give better performance than a single model if the models are somewhat independent of one another. A classifier that will give you the correct result 55% of the time is only mediocre, but if you've got 100 classifiers that are correct 55% of the time, the probability that the majority of them are right jumps to 82%. (Google "cumulative binomial probability" and try some numbers yourself.)

One way to get a variety of models that are somewhat independent from one another is to use different machine learning algorithms. For example, you can build models with support vector machine, linear regression, k nearest neighbors, binary decision tree, and so on. But it's difficult to get a very long list of models that way. And, besides, it's tedious because the different models all have different parameters that need to be tweaked separately and may have different requirements on the input data. So, the models need to be coded separately. That's not going to work for generating hundreds or thousands of models (which you'll be doing soon).

Therefore, the key with ensemble methods is to develop an algorithm approach to generate numerous somewhat independent models that will then be combined into an ensemble. In this chapter you learn how the most popular methods accomplish this. The chapter teaches you the mechanics of the most popular ensemble methods. It outlines the basic structure of the algorithms and demonstrates the algorithms in Python code to give you a firm understanding of their workings.

Ensemble methods employ a hierarchy of two algorithms. The low-level algorithm is called a *base learner*. The base learner is a single machine learning algorithm

that gets used for all of the models that will get aggregated into an ensemble. This chapter will primarily use binary decision trees as base learners. The upper-level algorithm manipulates the inputs to the base learners so that the models they generate are somewhat independent. How can the same algorithm generate different models? There are several upper-level algorithms that are widely used. They go by the names *bagging*, *boosting*, and *random forests*. (Strictly speaking, random forests is a combination of an upper-level algorithm and particular modification to binary decision trees. You will see more detail on that in the section "Random Forests").

A number of different algorithms could conceivably be used as base learners—binary decision trees, support vector machine, and so on—but as a practical matter binary decision trees are the most widely used. They are the base learners in the open source and commercial packages that you'll be able to use in your projects. The ensembles are collections of hundreds or thousands of binary decision trees, and many of the properties of these ensembles are ones they inherit from binary decision trees. So, this chapter begins with an introduction to binary decision trees.

Binary Decision Trees

Binary decision trees operate by subjecting attributes to a series of binary (yes/no) decisions. Each decision leads to one of two possibilities. Each decision leads to another decision or it leads to prediction. An example of a trained tree will help cement the idea. You'll learn how training works after understanding the result of training.

Listing 6-1 shows the code to use scikitlearn's DecisionTreeRegressor package to build a binary decision tree for the wine quality data. Figure 6-1 depicts the trained tree produced by Listing 6-1.

Listing 6-1: Building a Decision Tree to Predict Wine Quality—wineTree.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
from sklearn.externals.six import StringIO
from math import sqrt
import matplotlib.pyplot as plot

#read data into iterable
target_url = ("http://archive.ics.uci.edu/ml/machine-learning-"
"databases/wine-quality/winequality-red.csv")
data = urllib2.urlopen(target_url)

xList = []
```

```
labels = []
names = []
firstLine = True
for line in data:
    if firstLine:
        names = line.strip().split(";")
        firstLine = False
    else:
        #split on semi-colon
        row = line.strip().split(";")
        #put labels in separate array
        labels.append(float(row[-1]))
        #remove label from row
        row.pop()
        #convert row to floats
        floatRow = [float(num) for num in row]
        xList.append(floatRow)

nrows = len(xList)
ncols = len(xList[0])

wineTree = DecisionTreeRegressor(max_depth=3)

wineTree.fit(xList, labels)

with open("wineTree.dot", 'w') as f:
    f = tree.export_graphviz(wineTree, out_file=f)
#Note: The code above exports the trained tree info to a
#Graphviz "dot" file.
#Drawing the graph requires installing GraphViz and the running the
#following on the command line
#dot -Tpng wineTree.dot -o wineTree.png
# In Windows, you can also open the .dot file in the GraphViz
#gui (GVedit.exe)]
```

Figure 6-1 shows the series of decisions produced as an outcome of the training on the wine quality data. The block diagram of the trained tree shows a number of boxes, which are called *nodes* in decision tree parlance. There are two types of nodes: Nodes can either pose a yes/no question of the data, or they can be terminal nodes that assign a prediction to examples that end up in them. Terminal nodes are often referred to as *leaf* nodes. In Figure 6-1, the terminal nodes are the nodes at the bottom of the figure that have no branches or further decision nodes below them.

How a Binary Decision Tree Generates Predictions

When an observation or row is passed to a nonterminal node, the row answers the node's question. If it answers yes, the row of attributes is passed to the leaf node below and to the left of the current node. If the row answers no, the row of

attributes is passed to the leaf node below and to the right of the current node. The process continues recursively until the row arrives at a terminal (that is, leaf) node where a prediction value is assigned to the row. The value assigned by the leaf node is the mean of the outcomes of the all the training observations that wound up in the leaf node.

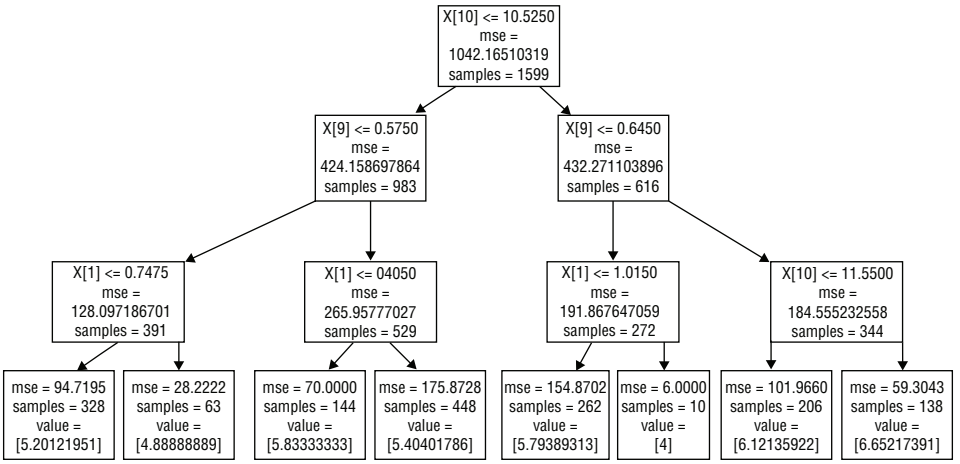

Figure 6-1: Decision tree for determining wine quality

While in this tree the second decision regards the variable X[9] in both branches of the tree, the two decisions can be about different attributes. (For example, see the third layer of decisions.)

Look at the top node, known as the *root* node. That node poses the question X[10] <= 10.525. In binary decision trees, important variables are split on early (or near the top of the tree), so the decision tree deems variable X[10], or alcohol content, important. In this respect, it agrees with the penalized linear regression in Chapter 5, "Building Predictive Models Using Penalized Linear Methods," which also deemed alcohol content most important in determining wine quality.

The tree in Figure 6-1 is said to have a depth of 3. The depth of a tree is defined as the number of decisions that have to be made down the longest path through the tree. The discussion of training in the section "Tree Training Equals Split Point Selection" will show you that there's no reason that all the paths to the terminal nodes have to be the same length (as they are in Figure 6-1).

You now have an idea what a trained tree looks like and you have seen how to how to use a trained tree to make predictions. Now you'll see how a tree gets trained.

How to Train a Binary Decision Tree

The easiest way to see how a tree gets trained is to look at a simple example. Listing 6-2 shows an example of predicting a real number label given a real

number attribute. The data set for this is created in the code (so called synthetic data). The basic idea is that the single attribute `x` has `100` equally spaced values between `-0.5` and `+0.5`. The labels `y` are equal to `x`, with some random noise added.

Listing 6-2: Training a Decision Tree for Simple Regression Problem—simpleTree.py

```
__author__ = 'mike-bowles'

import numpy
import matplotlib.pyplot as plot
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
from sklearn.externals.six import StringIO

#Build a simple data set with y = x + random
nPoints = 100

#x values for plotting
xPlot = [(float(i)/float(nPoints) - 0.5) for i in range(nPoints + 1)]

#x needs to be list of lists.
x = [[s] for s in xPlot]

#y (labels) has random noise added to x-value
#set seed
numpy.random.seed(1)
y = [s + numpy.random.normal(scale=0.1) for s in xPlot]

plot.plot(xPlot,y)
plot.axis('tight')
plot.xlabel('x')
plot.ylabel('y')
plot.show()

simpleTree = DecisionTreeRegressor(max_depth=1)
simpleTree.fit(x, y)

#draw the tree
with open("simpleTree.dot", 'w') as f:
    f = tree.export_graphviz(simpleTree, out_file=f)

#compare prediction from tree with true values

yHat  = simpleTree.predict(x)

plot.figure()
plot.plot(xPlot, y, label='True y')
plot.plot(xPlot, yHat, label='Tree Prediction ', linestyle='--')
```

continues

continued

```
plot.legend(bbox_to_anchor=(1,0.2))
plot.axis('tight')
plot.xlabel('x')
plot.ylabel('y')
plot.show()

simpleTree2 = DecisionTreeRegressor(max_depth=2)
simpleTree2.fit(x, y)

#draw the tree
with open("simpleTree2.dot", 'w') as f:
    f = tree.export_graphviz(simpleTree2, out_file=f)

#compare prediction from tree with true values

yHat  = simpleTree2.predict(x)

plot.figure()
plot.plot(xPlot, y, label='True y')
plot.plot(xPlot, yHat, label='Tree Prediction ', linestyle='--')
plot.legend(bbox_to_anchor=(1,0.2))
plot.axis('tight')
plot.xlabel('x')
plot.ylabel('y')
plot.show()

#split point calculations - try every possible split point to
#find the best one
sse = []
xMin = []
for i in range(1, len(xPlot)):
    #divide list into points on left and right of split point
    lhList = list(xPlot[0:i])
    rhList = list(xPlot[i:len(xPlot)])

    #calculate averages on each side
    lhAvg = sum(lhList) / len(lhList)
    rhAvg = sum(rhList) / len(rhList)

    #calculate sum square error on left, right and total
    lhSse = sum([(s - lhAvg) * (s - lhAvg) for s in lhList])
    rhSse = sum([(s - rhAvg) * (s - rhAvg) for s in rhList])

    #add sum of left and right to list of errors

    sse.append(lhSse + rhSse)
    xMin.append(max(lhList))

plot.plot(range(1, len(xPlot)), sse)
plot.xlabel('Split Point Index')
```

```
plot.ylabel('Sum Squared Error')
plot.show()

minSse = min(sse)
idxMin = sse.index(minSse)
print(xMin[idxMin])

#what happens if the depth is really high?
simpleTree6 = DecisionTreeRegressor(max_depth=6)
simpleTree6.fit(x, y)

#too many nodes to draw the tree
#with open("simpleTree2.dot", 'w') as f:
#    f = tree.export_graphviz(simpleTree6, out_file=f)

#compare prediction from tree with true values

yHat  = simpleTree6.predict(x)

plot.figure()
plot.plot(xPlot, y, label='True y')
plot.plot(xPlot, yHat, label='Tree Prediction ', linestyle='-')
plot.legend(bbox_to_anchor=(1,0.2))
plot.axis('tight')
plot.xlabel('x')
plot.ylabel('y')
plot.show()
```

Figure 6-2 plots the labels y versus the single attribute x. As you'd expect, y roughly follows x but with some randomness.

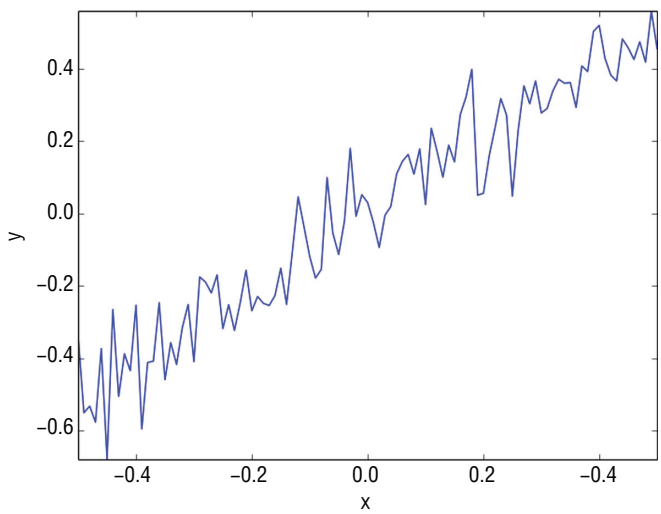

Figure 6-2: Label versus attribute for simple example

Tree Training Equals Split Point Selection

The first step in Listing 6-2 is to run scikitlearn's regression tree package with a depth of `1` specified. The results of that process are shown plotted in Figure 6-3. Figure 6-3 shows the block diagram for a depth 1 tree. Depth 1 trees are also called *stumps*. The single decision at the root node is to compare the attribute value with `-0.075`. This number is called the *split point* because it splits the data into two groups. The two boxes that emanate from the decision indicate that 43 of the 101 input examples go down the left leg of the tree, and the remaining 58 examples go down the right leg. If the attribute is less than the split point, the prediction from the tree is what's indicated as value in the block diagram—roughly –0.302.

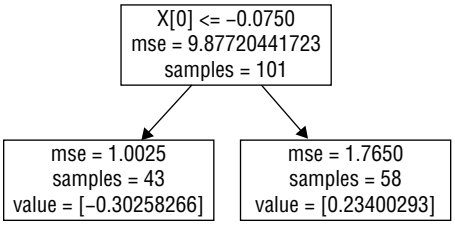

Figure 6-3: Block diagram of depth 1 tree for simple problem

How Split Point Selection Affects Predictions

Another way to view the trained tree is to see how its predictions compare with the true value of the labels. Because the simple synthetic problem has a single attribute only, the plot of the prediction generated by the trained tree alongside the actual values begins to give an idea about how the training of this simple tree was accomplished. The predicted values shown in Figure 6-4 follow a simple recipe. The prediction is a step function of the attribute. The step occurs at the split point.

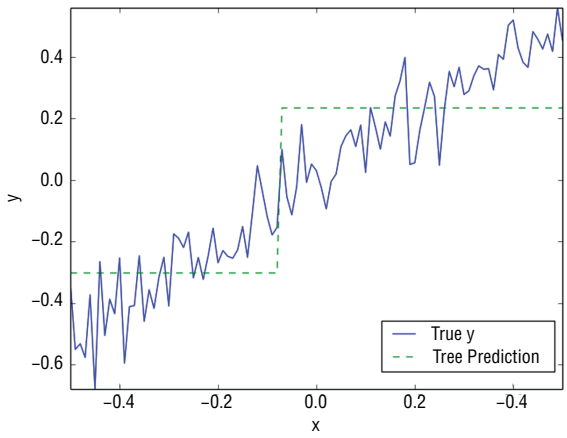

Figure 6-4: Comparison of predictions and actual values versus attribute for simple example

Algorithm for Selecting Split Points

Only three quantities are required to specify this simple tree: the split point value and the values assigned to the prediction if it falls into either of the two possible groups of points. Arriving at those quantities is accomplished during training of the tree. Here's how that works. The tree is trained to minimize the squared error of its predictions. Suppose first that the split point is given. Once the split point is given, the values assigned to the two groups are also determined. The average of each group is the single quantity that minimizes the mean squared error. That only leaves the question of how the split point is determined. Listing 6-2 has a small section of code that goes through the process of determining the split. The process is to try every possible split point. This is accomplished by dividing the data into two groups, approximating each group by its average, and then calculating the resulting sum squared error.

Figure 6-5 shows how the sum squared error varies as a function of the split point location. As you can see, there's a well-defined minimum at roughly the midpoint of the data set. Training a decision tree entails exhaustively searching all possible split points to determine which one minimizes the sum squared error. That takes care of this simple example.

Multivariable Tree Training—Which Attribute to Split?

What if the problem has more than one attribute? Then the algorithm checks all possible split points for all of the attributes to see which split point gives the best sum squared error for each attribute and then which attributes gives the overall minimum.

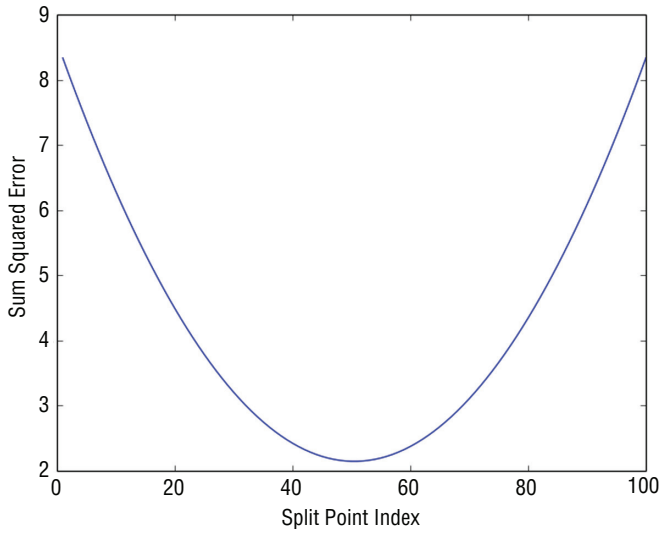

Figure 6-5: Sum squared error resulting from every possible split point location

This split point calculation is where all the computation cycles go in training a decision tree—and, by extension, where they go in training ensembles of trees. If the attribute being split doesn't have any repeat values, there's a split point to check for every data point (minus one).

As the data set gets larger, the number of split point calculations grows in direct proportion to the size of the data set. The split points that are checked can also get ridiculously close together. Algorithms designed to run on very large data sets allow split point checking to be considerably coarser than the raw granularity of the data. An approach to this is spelled out in "PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce,"[1] which outlines the approach taken by engineers at Google to build a decision tree algorithm on large data sets. As mentioned in the paper, they wanted the decision tree algorithm so that they could implement gradient boosting (one of the ensemble algorithms you'll learn about later in this chapter).

Recursive Splitting for More Tree Depth

Listing 6-2 shows what happens to the prediction curve as the tree depth increases from 1 to 2. The resulting prediction curves are shown in Figure 6-6, and the block diagram for the tree is shown in Figure 6-7. Instead of having a single step, the prediction curve now has three steps. The second set of split points is determined in the same manner as the first one. Each node in the tree deals with the subset of points determined by the splits above it. The split point for each node is determined to minimize the sum squared error in the two nodes below. The curve in Figure 6-6 approximates the actual curve with a finer stair-step function. More tree depth results in finer steps and higher fidelity to the training data. Will that continue indefinitely?

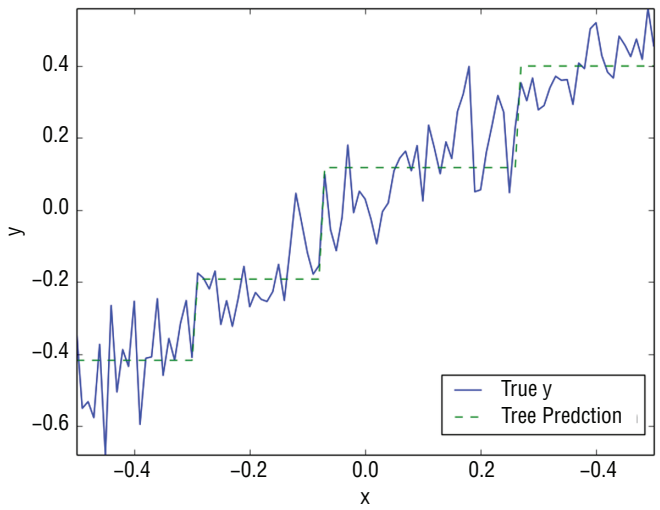

Figure 6-6: Prediction using depth 2 tree

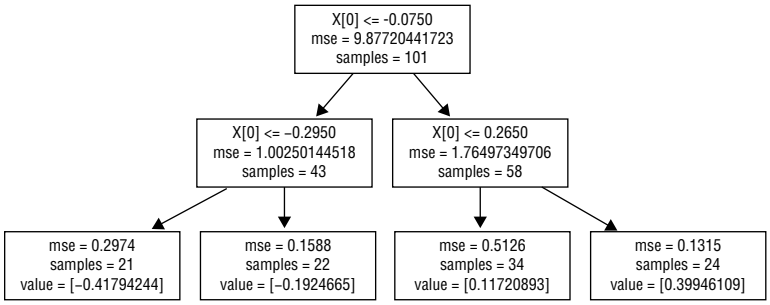

Figure 6-7: Block diagram for depth 2 tree

As splitting continues, the number of examples in the deepest nodes decreases. This can cause the splitting to terminate before the specified depth is reached. If there is only one example in a node, splitting certainly cannot continue. Tree training algorithms usually have a parameter to allow you to control how small a population will be split. Small populations in the nodes can lead to high variance in the resulting predictions.

Overfitting Binary Trees

The previous section showed how to train a binary decision tree of any depth. Is it possible to overfit a binary tree? This section discusses how to measure and regulate overfitting with binary trees. The mechanisms for overfitting binary trees are different from what you saw in Chapter 4, "Penalized Linear Regression," and Chapter 5, but you will see some similarities in the symptoms and how to measure overfitting. You will see that binary trees have parameters (tree depth and minimum leaf node size, for example) that can be used to regulate model complexity, similar to the process you saw in Chapters 4 and 5.

Measuring Overfit with Binary Trees

Figure 6-8 shows what happens when the tree depth is increased to 6. In Figure 6-8, it's hard to see the difference between the true value and the prediction. The prediction follows almost every zig and zag. That begins to suggest that the model is overfitting the data. The way the data were generated indicates that the best possible prediction would be for the prediction to equal the attribute value. The noise that was added to the attribute is unpredictable, and yet the prediction is following the noise-driven deviations of the label from the attribute. Synthetic data afford the luxury of knowing the correct answer.

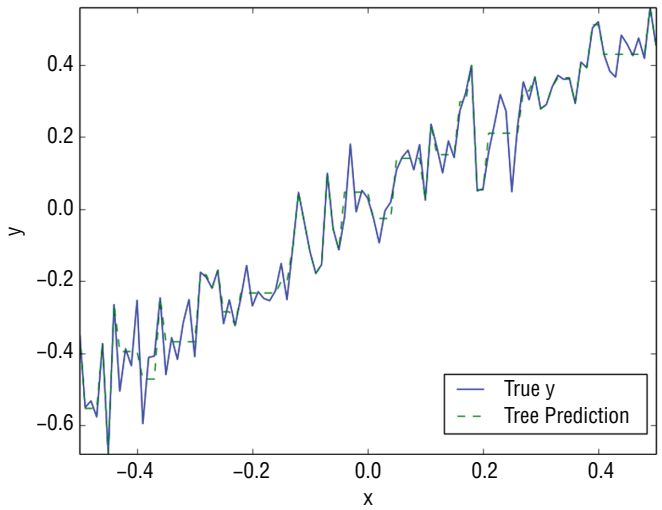

Figure 6-8: Prediction using depth 6 tree

Another way to look at overfitting with a binary tree is to consider the number of terminal nodes in the tree compared to the amount of data available. The tree that generated the prediction shown in Figure 6-8 was depth 6. That means that it has 64 terminal nodes (2^6). There are 100 points in the dataset. That means a lot of the points are the sole occupants of a terminal node, so their predicted value exactly matches their observed value. No wonder the graph of the prediction is matching the wiggles due to noise.

Balancing Binary Tree Complexity for Best Performance

In real problems, cross-validation can be performed to control overfitting. Listing 6-3 shows 10-fold cross-validation run on trees of a variety of depths for this simple problem. The code listing shows two loops. The outer one defines the tree depth for the inner cross-validation loop. The inner loop divides the data up and makes 10 passes to calculate out of sample errors. The mean squared error (MSE) results for each depth are plotted in Figure 6-9.

Listing 6-3: Cross-Validation at a Range of Tree Depths—simpleTreeCV.py

```
__author__ = 'mike-bowles'

import numpy
import matplotlib.pyplot as plot
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
from sklearn.externals.six import StringIO

#Build a simple data set with y = x + random
```

```
nPoints = 100

#x values for plotting
xPlot = [(float(i)/float(nPoints) - 0.5) for i in range(nPoints + 1)]

#x needs to be list of lists.
x = [[s] for s in xPlot]

#y (labels) has random noise added to x-value
#set seed
numpy.random.seed(1)
y = [s + numpy.random.normal(scale=0.1) for s in xPlot]

nrow = len(x)

#fit trees with several different values for depth and use
#x-validation to see which works best.

depthList = [1, 2, 3, 4, 5, 6, 7]
xvalMSE = []
nxval = 10

for iDepth in depthList:

    #build cross-validation loop to fit tree and evaluate on
    #out of sample data
    for ixval in range(nxval):

        #Define test and training index sets
        idxTest = [a for a in range(nrow) if a%nxval == ixval%nxval]
        idxTrain = [a for a in range(nrow) if a%nxval != ixval%nxval]

        #Define test and training attribute and label sets
        xTrain = [x[r] for r in idxTrain]
        xTest = [x[r] for r in idxTest]
        yTrain = [y[r] for r in idxTrain]
        yTest = [y[r] for r in idxTest]

        #train tree of appropriate depth and accumulate
        #out of sample (oos) errors
        treeModel = DecisionTreeRegressor(max_depth=iDepth)
        treeModel.fit(xTrain, yTrain)

        treePrediction = treeModel.predict(xTest)
        error = [yTest[r] - treePrediction[r] \
            for r in range(len(yTest))]

        #accumulate squared errors
        if ixval == 0:
            oosErrors = sum([e * e for e in error])
        else:
```

continues

continued

```
            #accumulate predictions
            oosErrors += sum([e * e for e in error])

    #average the squared errors and accumulate by tree depth

    mse = oosErrors/nrow
    xvalMSE.append(mse)

plot.plot(depthList, xvalMSE)
plot.axis('tight')
plot.xlabel('Tree Depth')
plot.ylabel('Mean Squared Error')
plot.show()
```

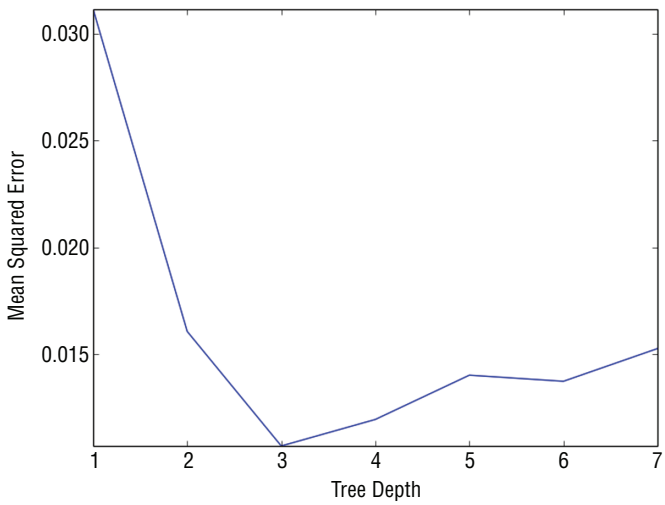

Figure 6-9: Out-of-sample error versus tree depth for simple problem

Tree depth is one way to regulate the complexity of a binary tree model. It has a similar effect to the coefficient penalty in the penalized regression model in Chapter 4 and Chapter 5. More tree depth makes it possible for the model to extract more complicated behaviors from the data at the cost of additional complexity. Figure 6-8 shows that depth 3 gives the best MSE performance for the synthetic problem from Listing 6-2. That depth makes the best trade off between reproducing the underlying relationships and overfitting the problem.

Recall from Chapter 3, "Predictive Model Building: Balancing Performance, Complexity, and Big Data," that the optimum model complexity is a function of the data set size. This synthetic data problem offers an opportunity to demonstrate how that works. Figure 6-10 shows how the optimum model complexity and performance change if the number of data points is increased to 1000.

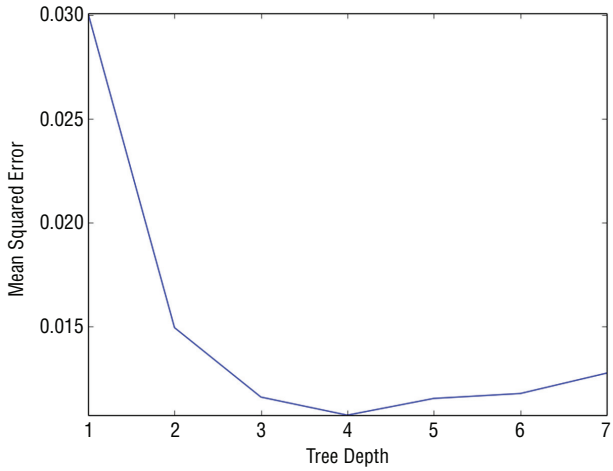

Figure 6-10: Out-of-sample MSE versus tree depth with 1,000 data points

You can run the plot for yourself by changing the variable `nPoints` in Listing 6-3 to `1000`. Two things happen as a result of adding more data. For one thing, the best tree depth increases from 3 to 4. The added data supports a more complicated model. For another thing, the MSE drops slightly. The added depth permits finer steps in the stair-step approximation of the real model. The added fidelity of the model is what excites people about really large data sets.

Modifications for Classification and Categorical Features

For you to have a complete picture of how decision trees are trained, there are a couple of other details to discuss. One is this: How does this work for a classification problem? The earlier criteria used to judge splits, MSE, makes sense only for regression problems. As you've seen elsewhere in the book, classification problems have different figures of merit than regression problems. Several figures of merit can be used with classification problems to judge splits in place of MSE. One that you're already familiar with is misclassification error. The other two commonly used measures are Gini impurity measure and information gain. For more information on these, see `http://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity`. These other two measures have somewhat different properties from misclassification error, but aren't conceptually different.

The last detail is how trees can be trained on attributes that are categorical instead of being numeric. The (nonterminal) nodes in the tree pose a yes/no question. For numeric variables, the question is in the form of whether the given attribute is less than a parameter. Splitting a categorical variable into two subsets consists of trying all the possible divisions of the categories into two sets. If the categories are A, B, and C, the possible splits are A in the first group and B and

C in the second, B in the first group and A and C in the second, and so forth. There are some mathematical results that simplify this in some circumstances.

This section furnished some background on binary decision trees. On their own, binary decision trees are a legitimate prediction tool and worthy of study, but the main purpose of outlining them here is as background for ensemble methods, which incorporate hordes of binary decision trees. You will see that some of the issues that come up using an individual tree (multiple parameters to adjust, structural instability, and overfitting for large trees) will recede into the background when the hundreds or thousands of these trees are combined. That was the intent behind the development of ensemble methods which are remarkably robust, easy to train, and accurate. The next sections discuss the three main ensemble methods one at a time.

Bootstrap Aggregation: "Bagging"

Bootstrap aggregation was developed by Leo Breiman.[2] This method starts with picking a base learner. The method will be implemented here using binary decision trees as the base learners. You'll see as we go through the method that other machine learning algorithms could be used as base learners. Binary decision trees are a logical choice because they naturally model problems with complicated decision boundaries, but binary decision trees can exhibit excessive performance variance. Variance can be overcome by combining a multitude of tree-based models.

How Does the Bagging Algorithm Work?

The bootstrap aggregation algorithm uses what is called a *bootstrap* sample. The bootstrap sample is often used for generating sample statistics from a modest data set. A (nonparametric) bootstrap sample is a random selection of several elements from the data set with replacement (that is, a bootstrap sample can contain multiple copies of a row from the original data). Bootstrap aggregation takes a number of bootstrap samples from the training data set and then trains a base learner on each of these samples. The resulting models are averaged in regression problems. For classification problems, the models can either be averaged or probabilities can be developed based on the percentages of different classes. Listing 6-4 shows code for the bagging algorithm applied to the synthetic problem introduced at the beginning of the chapter.

The code holds out 30% of the data for measuring out-of-sample performance instead of using cross-validation. The parameter `numTreesMax` determines the maximum number of trees that will be included in the ensemble. The code builds models from the first tree, the first two trees, the first three trees, and so on, up to `numTreesMax` trees, to see how the accuracy depends on the number of trees included in the ensemble. The code stores the trained models in a list and stores the predictions on the data that were held out for out-of-sample error testing.

The code produces two plots. One plots shows how the MSE changes as more trees are included in the ensemble. The second plot shows how the predictions from the first tree, the average of the first 10 trees and the average of the first 20 trees, compare. The comparison plot is similar to the plot of the prediction curve relative to the actual labels as functions of the single attribute.

Listing 6-4: Bootstrap Aggregation Algorithm—simpleBagging.py

```
__author__ = 'mike-bowles'

import numpy
import matplotlib.pyplot as plot
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
from math import floor
import random

#Build a simple data set with y = x + random
nPoints = 1000

#x values for plotting
xPlot = [(float(i)/float(nPoints) - 0.5) for i in range(nPoints + 1)]

#x needs to be list of lists.
x = [[s] for s in xPlot]

#y (labels) has random noise added to x-value
#set seed
random.seed(1)
y = [s + random.normal(scale=0.1) for s in xPlot]

#take fixed test set 30% of sample
nSample = int(nPoints * 0.30)
idxTest = random.sample(range(nPoints), nSample)
idxTest.sort()
idxTrain = [idx for idx in range(nPoints) if not(idx in idxTest)]

#Define test and training attribute and label sets
xTrain = [x[r] for r in idxTrain]
xTest = [x[r] for r in idxTest]
yTrain = [y[r] for r in idxTrain]
yTest = [y[r] for r in idxTest]

#train a series of models on random subsets of the training data
#collect the models in a list and check error of composite as list grows

#maximum number of models to generate
numTreesMax = 20
```

continues

continued

```
#tree depth - typically at the high end
treeDepth = 1

#initialize a list to hold models
modelList = []
predList = []

#number of samples to draw for stochastic bagging
nBagSamples = int(len(xTrain) * 0.5)

for iTrees in range(numTreesMax):
    idxBag = random.sample(range(len(xTrain)), nBagSamples)
    xTrainBag = [xTrain[i] for i in idxBag]
    yTrainBag = [yTrain[i] for i in idxBag]

    modelList.append(DecisionTreeRegressor(max_depth=treeDepth))
    modelList[-1].fit(xTrainBag, yTrainBag)

    #make prediction with latest model and add to list of predictions
    latestPrediction = modelList[-1].predict(xTest)
    predList.append(list(latestPrediction))


#build cumulative prediction from first "n" models
mse = []
allPredictions = []
for iModels in range(len(modelList)):

    #average first "iModels" of the predictions
    prediction = []
    for iPred in range(len(xTest)):
        prediction.append(sum([predList[i][iPred] \
            for i in range(iModels + 1)])/(iModels + 1))

    allPredictions.append(prediction)
    errors = [(yTest[i] - prediction[i]) for i in range(len(yTest))]
    mse.append(sum([e * e for e in errors]) / len(yTest))


nModels = [i + 1 for i in range(len(modelList))]

plot.plot(nModels,mse)
plot.axis('tight')
plot.xlabel('Number of Models in Ensemble')
plot.ylabel('Mean Squared Error')
plot.ylim((0.0, max(mse)))
plot.show()

plotList = [0, 9, 19]
for iPlot in plotList:
```

```
    plot.plot(xTest, allPredictions[iPlot])
plot.plot(xTest, yTest, linestyle="--")
plot.axis('tight')
plot.xlabel('x value')
plot.ylabel('Predictions')
plot.show()
```

Figure 6-11 shows how the MSE varies as the number of trees is increased. The error more or less levels out at around 0.025. This isn't really very good. The noise that was added had a standard deviation of 0.1. The very best MSE a predictive algorithm could generate is the square of that standard deviation or 0.01. The single binary tree that was trained earlier in the chapter was getting close to 0.01. Why is this more sophisticated algorithm underperforming?

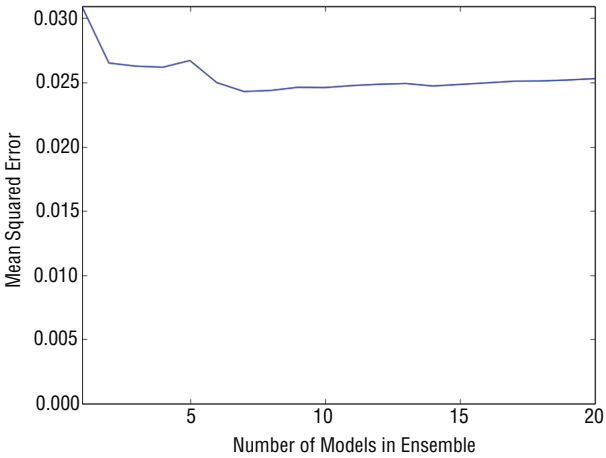

Figure 6-11: MSE versus number of trees in Bagging ensemble

Bagging Performance—Bias versus Variance

A look at Figure 6-12 gives some insight into the problem and raises a point that is important to illustrate because it's relevant to other problems too. Figure 6-12 shows the single tree prediction, the 10-tree prediction, and the 20-tree prediction. The prediction from the single tree is easy to discern because there's a single step. The 10- and 20-tree predictions superpose a number of slightly different trees so they have a series of finer steps that are in the neighborhood of the single step taken by the first tree. The steps of the multiple trees aren't all in exactly the same spot because they are trained on different samples of the data and that leads to some randomness in the split points. But that randomness only jiggles the split points in a relatively small neighborhood near the center of the graph. So, the resulting ensemble doesn't see much variety because all the trees in the ensemble roughly agree about where the single split point should go.

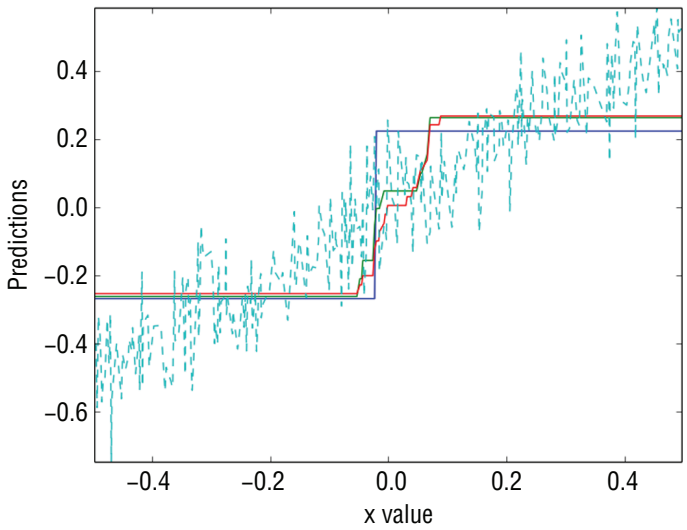

Figure 6-12: Comparison of prediction and actual label as functions of attribute

There are two types of error: bias and variance. Consider trying to fit a wiggly curve with a straight line. Getting more data can reduce the effect of noise in the data being used for fitting, but more data will not make a straight line into a wiggly curve. Errors that do not get smaller as more data points are added are called *bias errors*. Fitting depth-1 trees to the synthetic problem suffers from a bias error. All the split points are chosen near the center of the data, and the model accuracy suffers at the edges of the data.

The bias error with depth 1 trees comes from the basic model being too simple and sharing a common limitation. Bagging reduces variance between models. But with depth 1 trees, it gets a bias error, which can't be averaged. The way to overcome this problem is to use trees with more depth.

Figure 6-13 shows the curve of MSE versus number of trees in the ensemble for depth 5 trees. The MSE with depth 5 trees is somewhat smaller than 0.01 (probably due to randomness in the noise data), clearly much better performance than with depth 1 trees.

Figure 6-14 shows plots for the prediction based on the first tree, the first 10 trees, and the first 20 trees. The single tree prediction stands out from the others because it has a number of sharp spikes where it's making severe errors. In other words, it has a high variance. The other single trees undoubtedly show similar performance. But when they're average, the variance is reduced; the curve representing the prediction from the bagging algorithm is much smoother and closer to the true answer.

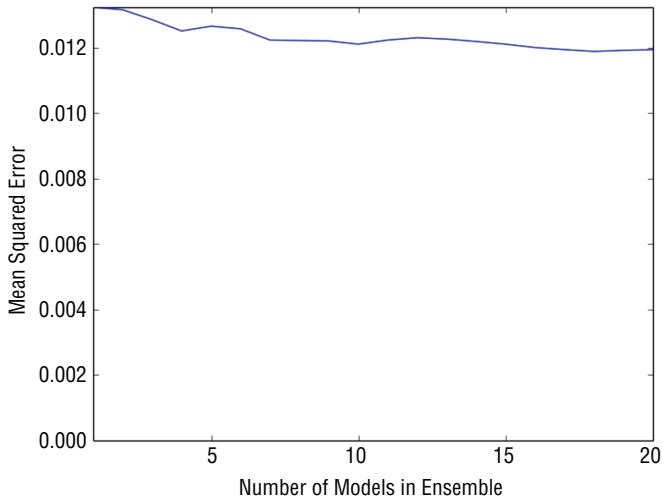

Figure 6-13: MSE versus number of trees with depth 5 trees

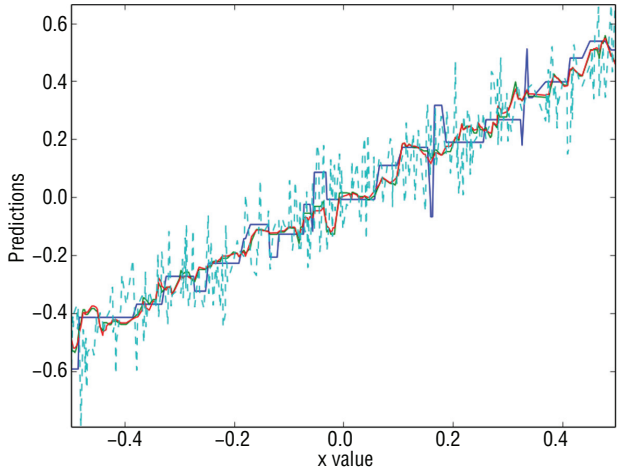

Figure 6-14: Comparison of prediction and actual labels with depth 5 trees

How Bagging Behaves on Multivariable Problem

Listing 6-5 shows the application of the bagging algorithm for the task of predicting wine quality. The wine example demonstrates some of the same principles as you saw with the synthetic data. These are best seen in Figures 6-15 through 6-17, which come from running Listing 6-4 with different parameter settings.

Listing 6-5: Predicting Wine Quality with Bagging—wineBagging.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
import random
from math import sqrt
import matplotlib.pyplot as plot

#read data into iterable
target_url = ("http://archive.ics.uci.edu/ml/machine-learning-"
"databases/wine-quality/winequality-red.csv")
data = urllib2.urlopen(target_url)

xList = []
labels = []
names = []
firstLine = True
for line in data:
    if firstLine:
        names = line.strip().split(";")
        firstLine = False
    else:
        #split on semi-colon
        row = line.strip().split(";")
        #put labels in separate array
        labels.append(float(row[-1]))
        #remove label from row
        row.pop()
        #convert row to floats
        floatRow = [float(num) for num in row]
        xList.append(floatRow)

nrows = len(xList)
ncols = len(xList[0])

#take fixed test set 30% of sample
random.seed(1)
nSample = int(nrows * 0.30)
idxTest = random.sample(range(nrows), nSample)
idxTest.sort()
idxTrain = [idx for idx in range(nrows) if not(idx in idxTest)]

#Define test and training attribute and label sets
xTrain = [xList[r] for r in idxTrain]
xTest = [xList[r] for r in idxTest]
yTrain = [labels[r] for r in idxTrain]
```

```
yTest = [labels[r] for r in idxTest]

#train a series of models on random subsets of the training data
#collect the models in a list and check error of composite as list grows

#maximum number of models to generate
numTreesMax = 30

#tree depth - typically at the high end
treeDepth = 1

#initialize a list to hold models
modelList = []
predList = []

#number of samples to draw for stochastic bagging
nBagSamples = int(len(xTrain) * 0.5)

for iTrees in range(numTreesMax):
    idxBag = []
    for i in range(nBagSamples):
        idxBag.append(random.choice(range(len(xTrain))))
    xTrainBag = [xTrain[i] for i in idxBag]
    yTrainBag = [yTrain[i] for i in idxBag]

    modelList.append(DecisionTreeRegressor(max_depth=treeDepth))
    modelList[-1].fit(xTrainBag, yTrainBag)

    #make prediction with latest model and add to list of predictions
    latestPrediction = modelList[-1].predict(xTest)
    predList.append(list(latestPrediction))


#build cumulative prediction from first "n" models
mse = []
allPredictions = []
for iModels in range(len(modelList)):

    #average first "iModels" of the predictions
    prediction = []
    for iPred in range(len(xTest)):
        prediction.append(sum([predList[i][iPred] \
            for i in range(iModels + 1)])/(iModels + 1))

    allPredictions.append(prediction)
    errors = [(yTest[i] - prediction[i]) for i in range(len(yTest))]
    mse.append(sum([e * e for e in errors]) / len(yTest))


nModels = [i + 1 for i in range(len(modelList))]
```

continues

continued

```
plot.plot(nModels,mse)
plot.axis('tight')
plot.xlabel('Number of Tree Models in Ensemble')
plot.ylabel('Mean Squared Error')
plot.ylim((0.0, max(mse)))
plot.show()

print('Minimum MSE')
print(min(mse))

#with treeDepth = 1
#Minimum MSE
#0.516236026081


#with treeDepth = 5
#Minimum MSE
#0.39815421341

#with treeDepth = 12 & numTreesMax = 100
#Minimum MSE
#0.350749027669
```

Figure 6-15 shows how MSE changes as more trees are included in the bagging ensemble. The ensemble of stumps (depth 1 trees) on the wine quality data shows negligible improvement in MSE over the single tree. The lack of improvement on the wine data is much more dramatic than with the synthetic data. This might be true for a couple of reasons. One possibility is that the errors at the edges of the data are more significant with the wine quality data than with the synthetic data. Another possibility is that interaction between variables is more important with the wine data.

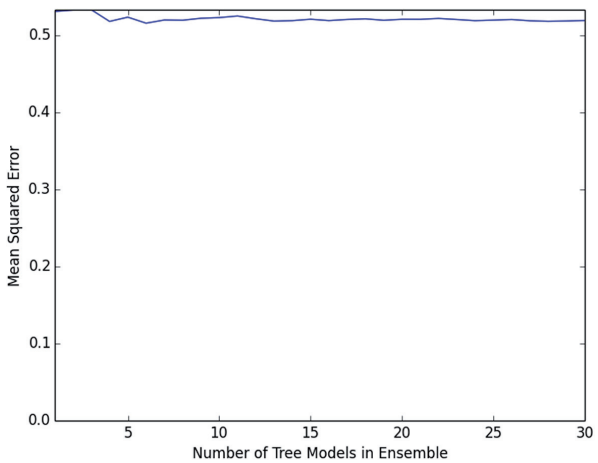

Figure 6-15: Predicting wine quality with Bagging on depth 1 trees

The synthetic data had only one variable, so no interaction between variables was possible. The wine data has multiple attributes, and so it's possible that the attributes in combination are more important than the sum of their individual contributions. If you stumble while walking, it won't likely be important. If you walk along the edge of a cliff, it won't likely be important. But if you stumble while walking along the edge of a cliff, it could be important. The two conditions have to be considered together. A depth 1 tree can only consider solitary attributes and therefore cannot account for strong interactions between variables.

Bagging Needs Tree Depth for Performance

Figure 6-16 shows how the MSE depends on number of trees when the trees all have depth 5. The Bagging ensemble shows clear improvement as more trees are added. The resulting performance is much better than that achieved by Bagging depth 1 trees. The improvement suggests that perhaps even more tree depth would yield further improvement.

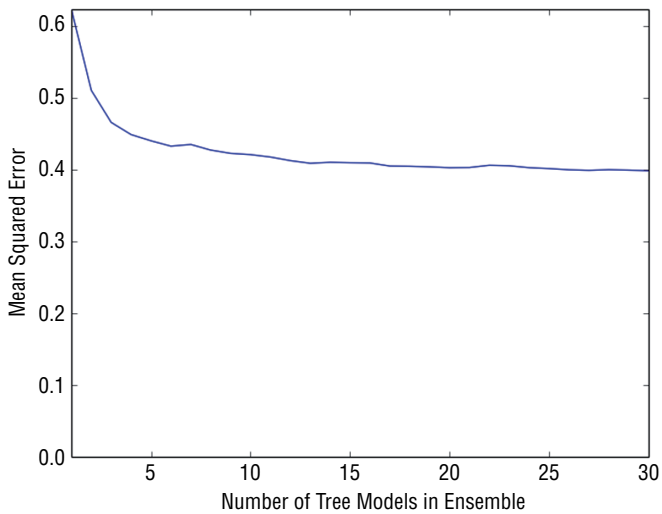

Figure 6-16: Predicting wine quality with Bagging on depth 5 trees

Figure 6-17 shows MSE versus number of trees in the Bagging ensemble when the trees are depth 12. In addition to employing deeper trees, the ensemble runs 100 trees rather than 30 to get a better picture of how much performance improvement is available by training larger numbers of trees for the Bagging ensemble. Figure 6-17 shows the lowest MSE of the three runs.

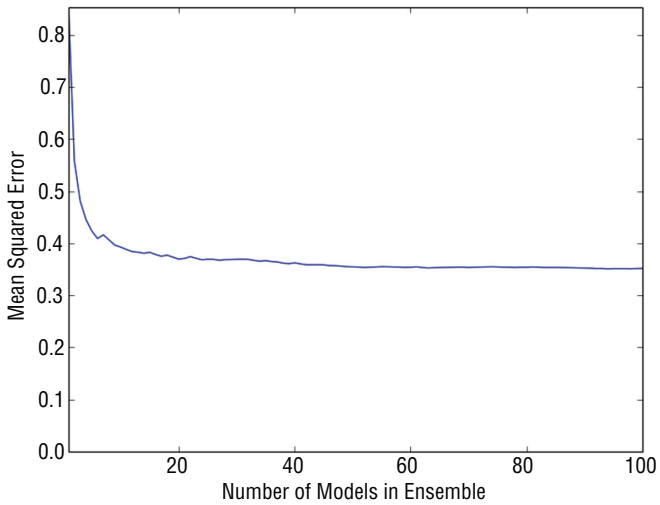

Figure 6-17: Predicting wine quality with Bagging on depth 12 trees

Summary of Bagging

Now you have seen a first example of an ensemble method. Bagging clearly demonstrates the two-level hierarchy common to ensemble methods. Properly speaking, bagging is the higher-level algorithm defining a series of subproblems to be solved by base learners and then averaging their predictions. The individual problems making up a bagging ensemble are derived by taking random bootstrap samples of the original training data. Bagging reduces the variance exhibited by individual binary trees. For bagging to work properly, the trees in a bagging ensemble need to be grown to sufficient depth.

Bagging serves as a good introduction to ensemble methods because it is relatively easy to understand and because it is relatively easy to demonstrate its variance reduction properties. The next two algorithms covered are gradient boosting and random forests. They take different approaches to building ensembles and exhibit some advantages over bagging. Most of the current practitioners I know use either gradient boosting or random forests first and do not regularly use bagging.

Gradient Boosting

Gradient boosting was developed by Stanford professor Jerome Friedman[4,5], who also developed the coordinate descent algorithm used to solve the ElasticNet problem (in Chapters 4 and 5). Gradient boosting develops an ensemble of tree-based models by training each of the trees in the ensemble on different labels and then combining the trees. For a regression problem where the objective is

to minimize MSE, each successive tree is trained on the errors left over by the collection of earlier trees. For the derivation of the algorithm see the References section at the end of this chapter. The easiest way to see how gradient boosting works is to look at some code implementing the algorithm.

Basic Principle of Gradient Boosting Algorithm

Listing 6-6 details the gradient boosting algorithm for the synthetic problem introduced earlier in this chapter. The early part of the code uses the process from earlier to build the synthetic data set.

Listing 6-6: Gradient Boosting for Simple Problem—simpleGBM.py

```
__author__ = 'mike-bowles'

import numpy
import matplotlib.pyplot as plot
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
from math import floor
import random

#Build a simple data set with y = x + random
nPoints = 1000

#x values for plotting
xPlot = [(float(i)/float(nPoints) - 0.5) for i in range(nPoints + 1)]

#x needs to be list of lists.
x = [[s] for s in xPlot]

#y (labels) has random noise added to x-value
#set seed
numpy.random.seed(1)
y = [s + numpy.random.normal(scale=0.1) for s in xPlot]

#take fixed test set 30% of sample
nSample = int(nPoints * 0.30)
idxTest = random.sample(range(nPoints), nSample)
idxTest.sort()
idxTrain = [idx for idx in range(nPoints) if not(idx in idxTest)]

#Define test and training attribute and label sets
xTrain = [x[r] for r in idxTrain]
xTest = [x[r] for r in idxTest]
yTrain = [y[r] for r in idxTrain]
yTest = [y[r] for r in idxTest]

#train a series of models on random subsets of the training data
#collect the models in a list and check error of composite as list grows
```

continues

continued

```
#maximum number of models to generate
numTreesMax = 30

#tree depth - typically at the high end
treeDepth = 5

#initialize a list to hold models
modelList = []
predList = []
eps = 0.3

#initialize residuals to be the labels y
residuals = list(yTrain)

for iTrees in range(numTreesMax):

    modelList.append(DecisionTreeRegressor(max_depth=treeDepth))
    modelList[-1].fit(xTrain, residuals)

    #make prediction with latest model and add to list of predictions
    latestInSamplePrediction = modelList[-1].predict(xTrain)

    #use new predictions to update residuals
    residuals = [residuals[i] - eps * latestInSamplePrediction[i] \
        for i in range(len(residuals))]

    latestOutSamplePrediction = modelList[-1].predict(xTest)
    predList.append(list(latestOutSamplePrediction))


#build cumulative prediction from first "n" models
mse = []
allPredictions = []
for iModels in range(len(modelList)):

    #add the first "iModels" of the predictions and multiply by eps
    prediction = []
    for iPred in range(len(xTest)):
        prediction.append(sum([predList[i][iPred]
            for i in range(iModels + 1)]) * eps)

    allPredictions.append(prediction)
    errors = [(yTest[i] - prediction[i]) for i in range(len(yTest))]
    mse.append(sum([e * e for e in errors]) / len(yTest))


nModels = [i + 1 for i in range(len(modelList))]

plot.plot(nModels,mse)
plot.axis('tight')
```

```
plot.xlabel('Number of Models in Ensemble')
plot.ylabel('Mean Squared Error')
plot.ylim((0.0, max(mse)))
plot.show()

plotList = [0, 14, 29]
lineType = [':', '-.', '--']
plot.figure()
for i in range(len(plotList)):
    iPlot = plotList[i]
    textLegend = 'Prediction with ' + str(iPlot) + ' Trees'
    plot.plot(xTest, allPredictions[iPlot], label = textLegend,
        linestyle = lineType[i])
plot.plot(xTest, yTest, label='True y Value', alpha=0.25)
plot.legend(bbox_to_anchor=(1,0.3))
plot.axis('tight')
plot.xlabel('x value')
plot.ylabel('Predictions')
plot.show()
```

Parameter Settings for Gradient Boosting

The first thing that looks unfamiliar is the comment about setting the depth parameter for the individual trees being trained in a gradient boosting ensemble. Gradient boosting differs from bagging and random forests in that it can reduce bias in addition to reducing variance. Gradient boosting has the useful property that it will often perform as well as low MSE values with stumps as with deeper trees. With gradient boosting, tree depth is only required to the extent that there's a significant interaction between variables. Performance improvement from increasing tree depth serves as a gauge of variable interaction in your problem.

The next thing that looks a little different is the definition of a variable called `eps`. This variable is a step size control of the sort that you may be familiar with from optimization problems. Gradient boosting takes gradient descent steps and, as with other gradient descent processes, if the steps are too large the optimization can diverge instead of converging. If the step size is too small, the process can take too many iterations. After generating some results, the chapter will talk about how to tune `eps`, the step size.

The next unfamiliar element of the code is the definition of a variable called `residuals`. The term *residuals* is commonly used to denote prediction errors (that is, observed values minus predicted values). The gradient boosting algorithm will make a series of refinements to its predictions of the labels. At each step along the way, the residuals will get recalculated. At the beginning of the process, gradient boosting initializes predictions to null (or zero) values so that the residuals are equal to the observed labels.

How Gradient Boosting Iterates Toward a Predictive Model

The loop on iTrees begins by training a tree using the attributes, but training on the residuals instead of the labels. Only for the first pass are the raw labels used for training targets. Subsequent passes take the predictions generated by training and subtract `eps` of them from the residuals before training. As mentioned, the subtraction of the residuals amount to a gradient descent and the reason for multiplying by the step size control parameter `eps` is to make sure that the iterative process converges. The code uses a fixed holdout set to measure out-of-sample performance and then plots the MSE as a function of the number of trees trained and also plots the function showing predicted values versus the single attribute.

Getting the Best Performance from Gradient Boosting

The first pair of plots (see Figures 6-18 and 6-19) shows the MSE versus number of trees and the plot of the prediction functions with eps = 0.1 and treeDepth = 1. Figure 6-18 shows that the error decreases smoothly and reaches roughly 0.014 after training 30 trees, and the MSE is heading down, indicating that it could be reduced further by training still more trees.

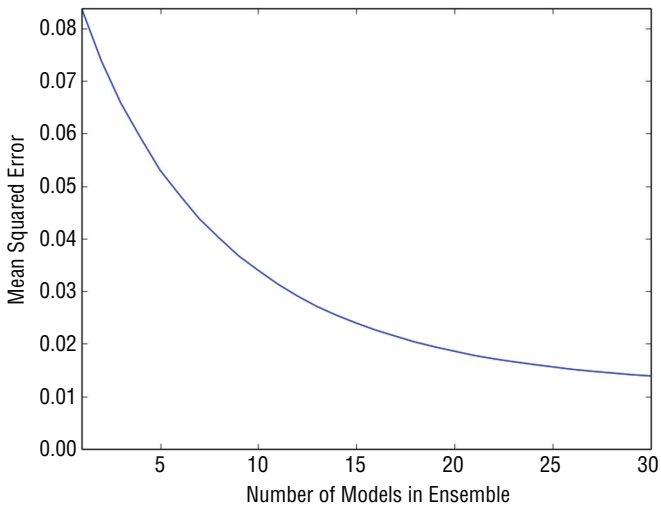

Figure 6-18: MSE versus number of trees for synthetic problem - eps = 0.1, treeDepth = 1

Figure 6-19 shows the prediction versus attribute value for three gradient boosting models—one that only trains one tree, one that trains 15 trees, and one that trains 30 trees. The model incorporating a single tree looks like a diminished version of the tree models that you saw in the introductory section about decision trees. As described, it is indeed a single depth 1 tree trained on the labels and then multiplied by 0.1—the value of `eps`. Things get more interesting with the

model built on 10 trees. That model makes a nice approximation to the correct answer—a straight line at 45 degrees on the graph. The model incorporating 10 trees correctly predicts roughly half of the range right, and predicts the right and left sides as constant. The model incorporating 30 trees gets a little further toward good approximation all the way to the edges of the data. This is distinct from the behavior that bagging showed with using stumps.

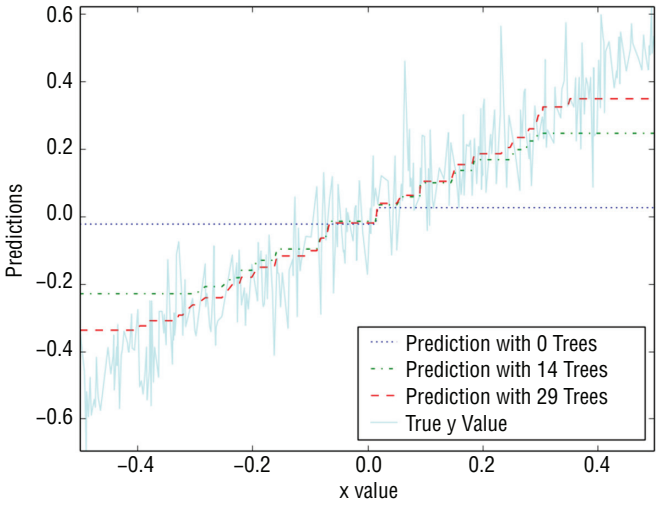

Figure 6-19: Gradient boosting predictions versus attribute value problem - eps = 0.1, treeDepth = 1

Bagging couldn't get beyond the bias error inherent in using shallow trees to build predictions for a number of problems not much different from one another. Gradient boosting starts in the same manner, but as it begins to reduce the errors in the middle of the data, it begins to pay more attention to the areas where it's making mistakes. That moves the split points out into the regions where there are mistakes. That process leads to a nice approximation without needing tree depth to get it.

What happens as the parameters controlling the training are changed? Figures 6-20 and 6-21 show how the picture changes if trees are of depth 5. The MSE plot in Figure 6-20 shows a similar smooth reduction in MSE as the number of trees increases. The MSE value gets very close to perfection (0.01) after training 30 depth 5 trees—lower than with depth 1 trees. What the plot doesn't show is training time. Each level in a tree takes about the same time to train. At each layer, all the possible split points have to be compared for MSE. A depth 5 tree takes five times as long as five depth 1 trees. A fair comparison would be to see what error the depth 1 trees reached after 150 trees compared to depth 5 trees after 30.

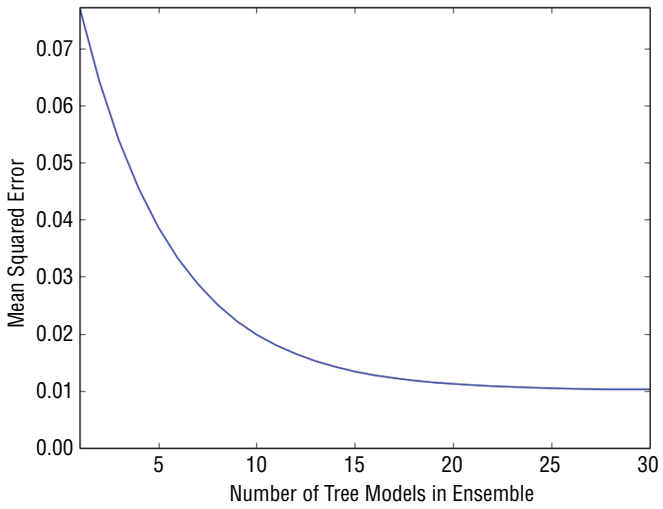

Figure 6-20: MSE versus number of trees for synthetic problem - eps = 0.1, treeDepth = 5

Figure 6-21 clearly reflects the deeper trees being used to build the gradient boosting ensemble. Even the first prediction based on a single tree shows some structure all across the range of the attribute. The models based on 15 trees and 30 trees still exhibit higher levels of error at the edges of the data.

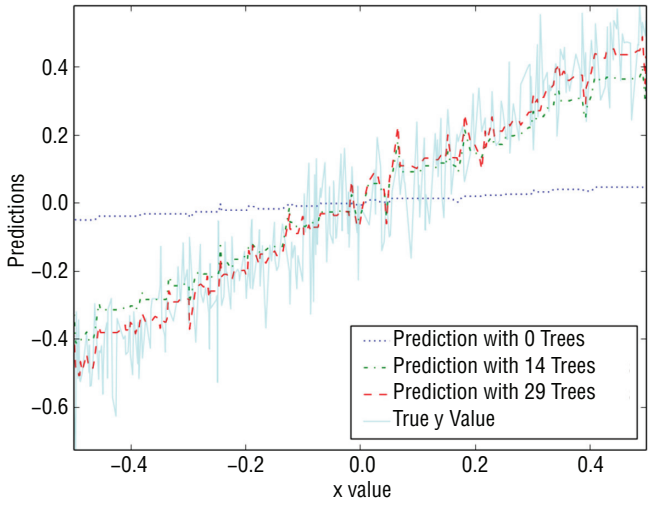

Figure 6-21: Gradient boosting predictions versus attribute value problem - eps = 0.1, treeDepth = 5

Figures 6-22 and 6-23 show what happens as the step size parameter `eps` is increased. Figure 6-22 shows behavior that's characteristic of too large a step size parameter (named `eps` here). The graph of MSE versus the number of trees decreases

sharply but then increases again toward the right side. The minimum is on the left side of the graph, near the one-third point. You want to adjust `eps` so that the minimum is at or near the right edge of the graph. That usually gives better performance.

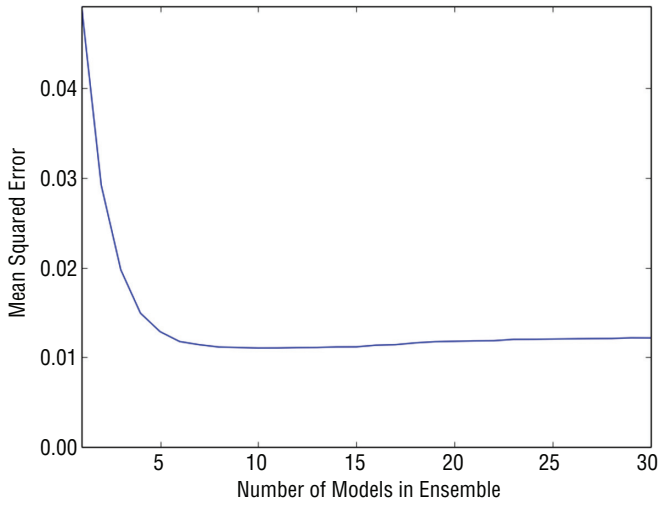

Figure 6-22: MSE versus number of trees for synthetic problem - eps = 0.3, treeDepth = 5

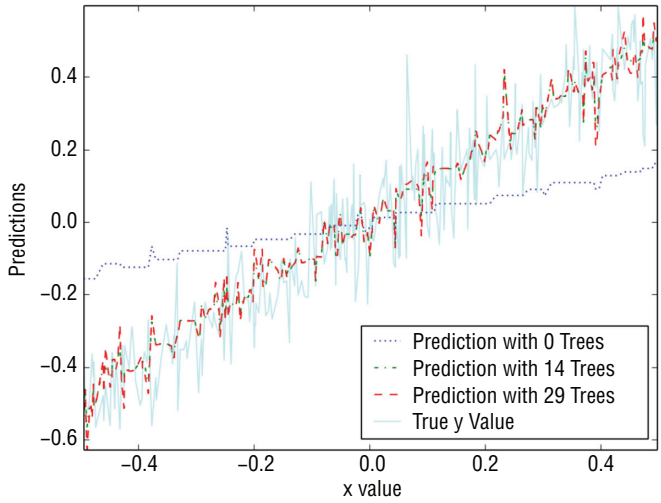

Figure 6-23: Gradient Boosting predictions versus attribute value problem - eps = 0.3, treeDepth = 5

The picture of the predictions as a function of the attribute shows more spiky diversions from the correct 45% line than either the versions using *eps=0.1*. Overall, the version with depth 1 trees is the best behaved. It looks like training more trees might improve the performance at the edges of the depth 1 model and lead to the best answer for gradient boosting.

Gradient Boosting on a Multivariable Problem

Listing 6-7 shows application of gradient boosting to the task of predicting wine quality. With the exception using the wine data set for input, the code in Listing 6-6 is very similar to the code used on the simple synthetic data set.

Listing 6-7: Gradient Boosting for Prediction Wine Quality—wineGBM.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
import random
from math import sqrt
import matplotlib.pyplot as plot

#read data into iterable
target_url = "http://archive.ics.uci.edu/ml/machine-learning-"
"databases/wine-quality/winequality-red.csv")
data = urllib2.urlopen(target_url)

xList = []
labels = []
names = []
firstLine = True
for line in data:
    if firstLine:
        names = line.strip().split(";")
        firstLine = False
    else:
        #split on semi-colon
        row = line.strip().split(";")
        #put labels in separate array
        labels.append(float(row[-1]))
        #remove label from row
        row.pop()
        #convert row to floats
        floatRow = [float(num) for num in row]
        xList.append(floatRow)

nrows = len(xList)
ncols = len(xList[0])

#take fixed test set 30% of sample
nSample = int(nrows * 0.30)
idxTest = random.sample(range(nrows), nSample)
idxTest.sort()
idxTrain = [idx for idx in range(nrows) if not(idx in idxTest)]
```

```
#Define test and training attribute and label sets
xTrain = [xList[r] for r in idxTrain]
xTest = [xList[r] for r in idxTest]
yTrain = [labels[r] for r in idxTrain]
yTest = [labels[r] for r in idxTest]

#train a series of models on random subsets of the training data
#collect the models in a list and check error of composite as list grows

#maximum number of models to generate
numTreesMax = 30

#tree depth - typically at the high end
treeDepth = 5

#initialize a list to hold models
modelList = []
predList = []
eps = 0.1

#initialize residuals to be the labels y
residuals = list(yTrain)

for iTrees in range(numTreesMax):

    modelList.append(DecisionTreeRegressor(max_depth=treeDepth))
    modelList[-1].fit(xTrain, residuals)

    #make prediction with latest model and add to list of predictions
    latestInSamplePrediction = modelList[-1].predict(xTrain)

    #use new predictions to update residuals
    residuals = [residuals[i] - eps * latestInSamplePrediction[i] \
        for i in range(len(residuals))]

    latestOutSamplePrediction = modelList[-1].predict(xTest)
    predList.append(list(latestOutSamplePrediction))


#build cumulative prediction from first "n" models
mse = []
allPredictions = []
for iModels in range(len(modelList)):

    #add the first "iModels" of the predictions and multiply by eps
    prediction = []
    for iPred in range(len(xTest)):
        prediction.append(sum([predList[i][iPred]
            for i in range(iModels + 1)]) * eps)

    allPredictions.append(prediction)
```

continues

continued

```
    errors = [(yTest[i] - prediction[i]) for i in range(len(yTest))]
    mse.append(sum([e * e for e in errors]) / len(yTest))

nModels = [i + 1 for i in range(len(modelList))]

plot.plot(nModels,mse)
plot.axis('tight')
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Mean Squared Error')
plot.ylim((0.0, max(mse)))
plot.show()

print('Minimum MSE')
print(min(mse))

#printed output
#Minimum MSE
#0.405031864814
```

The parameter selections shown in the code are for 30 depth 5 trees and *eps=0.1*. This parameter set yields MSE of roughly 0.4. That's about 10% worse than the performance bagging got on the same problem. Try adjusting the number of trees, `eps`, the step size parameter, and the tree depth to see whether you can get better results.

The curve of MSE versus number of trees looks fairly flat at the right edge (see Figure 6-24). It might still be possible to get some more performance by adding more trees to the ensemble. The other possible approaches to squeezing out a little more performance would be to tweak the tree depth or step size parameter.

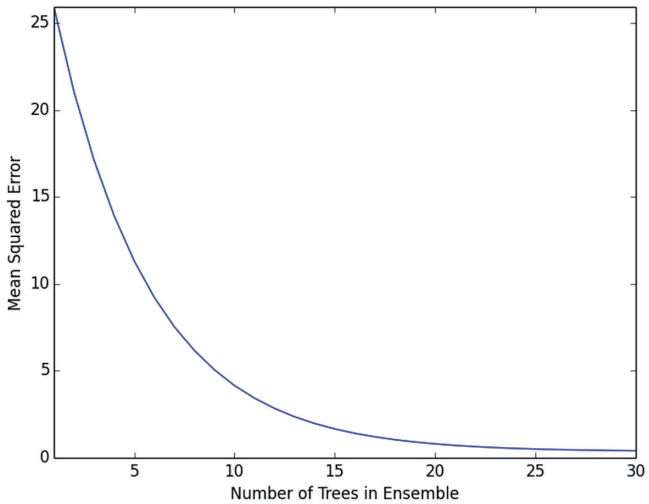

Figure 6-24: MSE versus number of trees for Gradient Boosting model of wine quality

Summary for Gradient Boosting

This section has shown how gradient boosting operates and demonstrated how to control its behavior to get the best performance. The section talked about the effect of changing step size, tree depth, and number of trees. You've seen how gradient boosting avoids the bias errors that bagging experienced with shallow trees. The basic difference in principle between bagging and boosting is that boosting constantly monitors its cumulative error and uses that residual for subsequent training. That difference accounts for gradient boosting only needing tree depth when there's significant interaction among the various attributes in the problem.

Random Forest

The random forests algorithm was developed by the late Berkeley professor Leo Breiman and Adele Cutler.[3] Random forests generates its sequence of models by training them on subsets of the data. The subsets are drawn at random from the full training set. One way in which the subset is selected is to randomly sample rows with replacement in the same manner as Brieman's bootstrap aggregation algorithm. The other random element is that the training sets for the individual trees in the random forests ensemble don't incorporate all the attributes but take a random subset of the attributes also. Listing 6-8 approximates random forests using Python DecisionTreeRegression.

Listing 6-8: Bagging with Random Attribute Selection—wineRF.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor
import random
from math import sqrt
import matplotlib.pyplot as plot

#read data into iterable
target_url = ("http://archive.ics.uci.edu/ml/machine-learning-"
"databases/wine-quality/winequality-red.csv")
data = urllib2.urlopen(target_url)

xList = []
labels = []
names = []
firstLine = True
for line in data:
    if firstLine:
```

continues

continued

```
        names = line.strip().split(";")
        firstLine = False
    else:
        #split on semi-colon
        row = line.strip().split(";")
        #put labels in separate array
        labels.append(float(row[-1]))
        #remove label from row
        row.pop()
        #convert row to floats
        floatRow = [float(num) for num in row]
        xList.append(floatRow)

nrows = len(xList)
ncols = len(xList[0])

#take fixed test set 30% of sample
random.seed(1)  #set seed so results are the same each run
nSample = int(nrows * 0.30)
idxTest = random.sample(range(nrows), nSample)
idxTest.sort()
idxTrain = [idx for idx in range(nrows) if not(idx in idxTest)]

#Define test and training attribute and label sets
xTrain = [xList[r] for r in idxTrain]
xTest = [xList[r] for r in idxTest]
yTrain = [labels[r] for r in idxTrain]
yTest = [labels[r] for r in idxTest]

#train a series of models on random subsets of the training data
#collect the models in a list and check error of composite as list grows

#maximum number of models to generate
numTreesMax = 30

#tree depth - typically at the high end
treeDepth = 12

#pick how many attributes will be used in each model.
# authors recommend 1/3 for regression problem
nAttr = 4

#initialize a list to hold models
modelList = []
indexList = []
predList = []
nTrainRows = len(yTrain)

for iTrees in range(numTreesMax):

    modelList.append(DecisionTreeRegressor(max_depth=treeDepth))
```

```
    #take random sample of attributes
    idxAttr = random.sample(range(ncols), nAttr)
    idxAttr.sort()
    indexList.append(idxAttr)

    #take a random sample of training rows
    idxRows = []
    for i in range(int(0.5 * nTrainRows)):
        idxRows.append(random.choice(range(len(xTrain))))
    idxRows.sort()

    #build training set
    xRfTrain = []
    yRfTrain = []

    for i in range(len(idxRows)):
        temp = [xTrain[idxRows[i]][j] for j in idxAttr]
        xRfTrain.append(temp)
        yRfTrain.append(yTrain[idxRows[i]])

    modelList[-1].fit(xRfTrain, yRfTrain)

    #restrict xTest to attributes selected for training
    xRfTest = []
    for xx in xTest:
        temp = [xx[i] for i in idxAttr]
        xRfTest.append(temp)

    latestOutSamplePrediction = modelList[-1].predict(xRfTest)
    predList.append(list(latestOutSamplePrediction))


#build cumulative prediction from first "n" models
mse = []
allPredictions = []
for iModels in range(len(modelList)):

    #add the first "iModels" of the predictions and multiply by eps
    prediction = []
    for iPred in range(len(xTest)):
        prediction.append(sum([predList[i][iPred]
            for i in range(iModels + 1)]) / (iModels + 1))

    allPredictions.append(prediction)
    errors = [(yTest[i] - prediction[i]) for i in range(len(yTest))]
    mse.append(sum([e * e for e in errors]) / len(yTest))


nModels = [i + 1 for i in range(len(modelList))]

plot.plot(nModels,mse)
```

continues

continued

```
plot.axis('tight')
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Mean Squared Error')
plot.ylim((0.0, max(mse)))
plot.show()

print('Minimum MSE')
print(min(mse))

#printed output

#Depth 1
#Minimum MSE
#0.52666715461

#Depth 5
#Minimum MSE
#0.426116327584

#Depth 12
#Minimum MSE
#0.38508387863
```

Random Forests: Bagging Plus Random Attribute Subsets

The example shown in Listing 6-5 trains on the wine quality data set. The simple single-attribute example that was used earlier to illustrate bagging and gradient boosting algorithms won't work with random forests. That example had only one attribute. It does not make sense to take a random draw of a single item. The code in Listing 6-5 looks a lot like the code for bagging. The only difference between the two that shows up before the loop on iTrees is the specification of a variable called `nAttr`. The random draw on the attributes needs to know how many attributes to select. The authors of the original paper recommend one third the number of attributes for a regression problem (and the square root of the number of attributes for a classification problem). Inside the iTrees loop, there's a random sample on rows of the attribute matrix—just like with bagging. There's also a random draw without replacement on the columns of the attribute matrix (or what would be rows and columns if the list of lists were converted to a numpy array). Then a tree gets trained and used to make a prediction on the out-of-sample data.

There is a difference between what's implemented in Listing 6-5 and the random forests algorithm. The algorithm in Listing 6-5 takes a random subset of the attributes and trains a tree with that subset. Breiman's original version of the random forests algorithm takes a different random set of attributes for each node in the tree. To implement Breiman's original version of the algorithm requires access to the innards of the tree growing algorithm. The example, nonetheless, gives a feel for how the algorithm operates. Some people argue that there's not much advantage to make the random draw on attributes at every node.

Random Forests Performance Drivers

Figures 6-25 through 6-27 show how the addition of random attribute selection affect the curves of MSE versus the number of trees included in the ensemble. Figure 6-25 shows the result when the individual trees are depth 1 trees. The picture is very similar to bagging in that the ensemble doesn't improve performance very much. Depth 1 trees mostly cause bias error not variance error. Bias can't be averaged away.

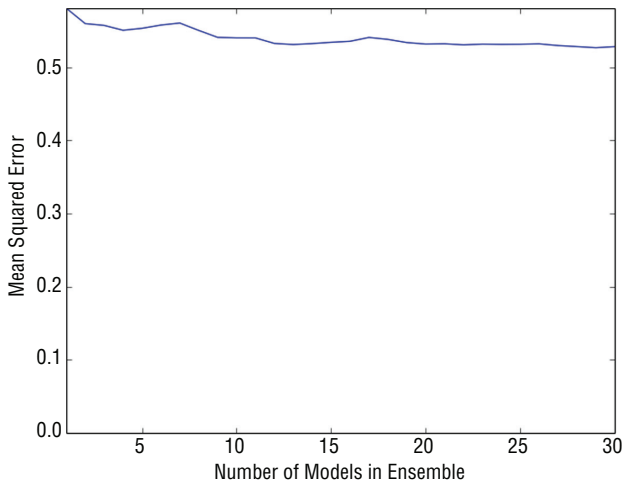

Figure 6-25: MSE versus number of trees for bagging + random attribute selection – Depth 1 trees

Figure 6-26 shows the MSE curve using depth 5 trees. Now the variance reduction with bagging plus random attribute selection begins to show some performance. The improvement with this combination gets similar performance to other methods demonstrated.

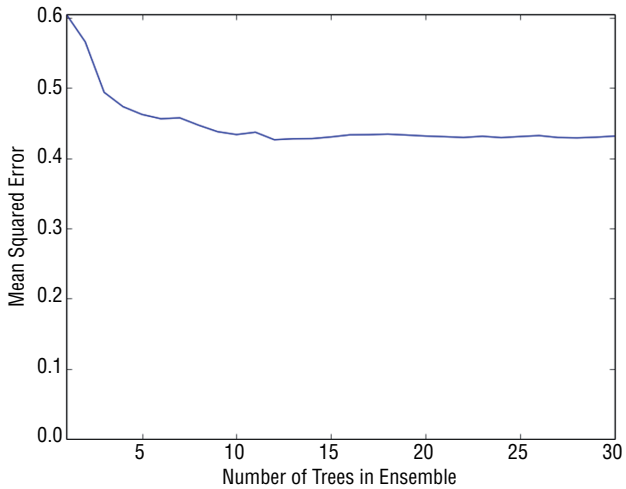

Figure 6-26: MSE versus number of trees for bagging + random attribute selection – Depth 5 trees

Figure 6-27 shows that a little more performance is available by using depth-12 trees.

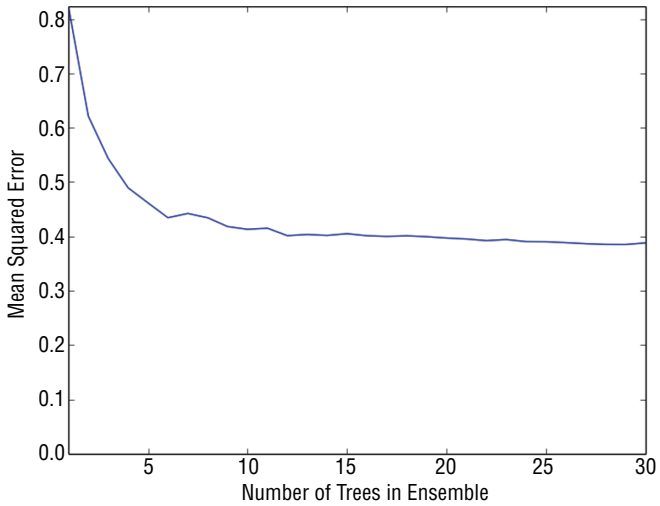

Figure 6-27: MSE versus number of trees for bagging + random attribute selection – Depth 12 trees

Random Forests Summary

Random forests is a combination of bagging and a random attribute selection modification to the binary tree base learners. These differences may not seem substantial, but they give random forests different performance characteristics from bagging and gradient boosting. Some results suggest that random forests has an advantage with wide sparse attribute spaces such as occur in text mining problems. Random forests is a little easier to parallelize than gradient boosting because the individual base learners can be trained independently of one another whereas with gradient boosting each base learner needs the results from the ones before it.

Differences like these mean that you may want to try both random forests in addition to gradient boosting, if you need to wring as much performance as possible from your data.

Summary

This chapter gave you some background on basic ensemble algorithms. It explained that ensemble methods consisted of a hierarchy of two algorithms. Ensemble methods train hundreds or thousands of the low-level algorithms called *base learners*. The higher-level algorithm controls the training of the base learners in order that their models turn out somewhat independent from one

another so that combining them will reduce the variance of the combination. For bagging, the higher-level algorithm is to take bootstrap samples of the training set and train base learners on these samples. For gradient boosting, the higher-level algorithm at each stage takes a sample of the input data and trains a base learner on it. With gradient boosting, the target used to train each base learner is the error from the accumulation of all the earlier base learners. Random forests is a combination of bagging as a higher-level algorithm and base learners that are modified versions of binary decision trees. The base learners with random forests are binary trees where, at each node, the split point decisions are restricted to a random sample of the available attributes instead of considering all the attributes in each split. The packages available for doing gradient boosting in Python permit you to use random forests base learners with gradient boosting. You will see that use in the next chapter, "Building Ensemble Methods with Python."

The chapter coded each of the high-level algorithms and showed a facsimile of the random forests base learners. The purpose for coding these is for you to gain an understanding of the mechanisms at work in each of the algorithms. The idea behind that is that you will better understand the options, input variables, nominal starting values, and so on for the Python packages for these algorithms. The next chapter uses available Python packages to generate solutions to some of the problems you've seen solved by penalized linear regression.

References

1. Panda Biswanath, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. (2009). PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. Proceedings of the 35th International Conference on Very Large Data Bases. Retrieved from `http://research.google.com/pubs/pub36296.html`.
2. Leo Breiman. (September, 1994). Bagging Predictors. Technical Report No. 421. Department of Statistics, UC Berkeley. Retrieved from `http://statistics.berkeley.edu/sites/default/files/tech-reports/421.pdf`.
3. Leo Breiman. (2001). Random forests. *Machine Learning*, *45*:5–32. Retrieved from `http://oz.berkeley.edu/~breiman/randomforest2001.pdf`.
4. J.H. Friedman. (2001). Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics*, *29*(5):1189–1232. Retrieved from `http://statweb.stanford.edu/~jhf/ftp/trebst.pdf`.
5. J.H. Friedman. (2002). Stochastic Gradient Boosting. *Computational Statistics and Data Analysis*, *38*(4):367–378. Retrieved from `http://statweb.stanford.edu/~jhf/ftp/stobst.pdf`.