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

CHAPTER
7

Building Ensemble Models with Python

This chapter uses several available Python packages to build predictive models using the ensemble algorithms that you saw in Chapter 6, "Ensemble Methods." The problems used to illustrate them were introduced in Chapter 2, "Understand the Problem by Understanding the Data." You saw in Chapter 5, "Building Predictive Models Using Penalized Linear Methods," how to build predictive models for them using penalized linear regression. This chapter uses ensemble methods to solve the same problems. That will enable you to compare the algorithms and the available Python packages in terms of how easy the packages are to use, what kinds of accuracy is achievable with ensemble methods versus penalized linear regression, how the training times compare, and so on. The end of the chapter shows some summary comparisons of the various algorithms you've become familiar with.

Solving Regression Problems with Python Ensemble Packages

The next several sections demonstrate the application of available Python packages for building ensemble models. You will see the things you learned in Chapter 6 in action. The methods explained in Chapter 6 will be used on the series of problems explored in Chapter 2 and then used to demonstrate the application of penalized linear regression in Chapter 5. Using the same problems makes it possible to compare the algorithms covered here along several dimensions, including raw performance, training time, and ease of use. The chapter also covers the available Python packages. The background given in Chapter 6 helps you understand why the Python packages are structured

the way they are and helps you see how to get the most from these methods. This section goes through a variety of different problem types, beginning with regression problems.

Building a Random Forest Model to Predict Wine Taste

The wine quality data set provides an opportunity to predict wine taste scores based on the chemical composition of wine. As you know by now, this problem type is called a regression problem because the predictions take the form of real numbers. The Python scikit-learn ensemble module houses a Random Forest algorithm and a Gradient Boosting algorithm, both of which are for regression problems. First, this section explains the parameters required to instantiate a member of the `RandomForestRegressor` class. Then this section uses the `RandomForestRegressor` class to train a Random Forests model for the wine taste data and to explore the performance of the model.

Constructing a RandomForestRegressor Object[1]

Here is the class constructor for `sklearn.ensemble.RandomForestRegressor`:

```
sklearn.ensemble.RandomForestRegressor(n_estimators=10, criterion='mse',
max_depth=None, min_samples_split=2, min_samples_leaf=1, max_features=
'auto', max_leaf_nodes=None, bootstrap=True, oob_score=False, n_jobs=1,
random_state=None, verbose=0, min_density=None, compute_importances=
None)
```

The following description mirrors sklearn documentation, but covers only the parameter values that you're most likely to want to alter.[1] For those parameters, the list describes how to choose alternatives to the default values. To see descriptions of the parameters not covered here, see the sklearn package documentation. The following list describes the parameters:

- **n_estimators**

  integer, optional (default = 10)

  This is the number of trees in the ensemble. The default is okay to use if you coded things correctly, but you'll generally want more than 10 trees to gain the best performance. You can experiment with the number and get a feel for how many are required. As emphasized throughout this book, the appropriate model complexity (tree depth and number of trees) depends on the complexity of the underlying problem and the amount of data that you have. A good starting point is 100–500.

- **max_depth**

  integer or None, optional (default=None)

  If this parameter is set to `None`, the tree will be grown until all the leaf nodes are either pure or they hold fewer than `min_samples_split` examples. As an alternative to specifying the tree depth, you can use `max_leaf_nodes` to specify the number of leaf nodes in the tree. If you specify `max_leaf_nodes`, `max_depth` is ignored. There might be a performance advantage to leaving `max_depth` set to auto and growing full-depth trees. This is also a training time cost associated with full-depth trees. You may want to experiment with the depth if you need several training runs to complete your modeling process.

- **min_samples_split**

  integer, optional (default=2)

  Nodes will not be split that have fewer than `min_samples_split` examples. Splitting nodes that are small is a source of overfitting.

- **min_samples_leaf**

  integer, optional (default=1)

  A split is not taken if the split leads to nodes that have fewer than `min_samples_leaf`. The default value for this parameter results in the parameter being ignored, which is often okay—particularly when you're making the first few training runs on your data set. You can think about selecting a meaningful value for this parameter in a couple of ways. One is that the value assigned to a leaf is the average of the examples in the leaf and that you'll get a lower variance average if there's more than one sample in the leaf node. Another way to think about this parameter is as an alternative way to control tree depth.

- **max_features**

  integer, float or string, optional (default=None)

  The number of features to consider when looking for the best split depends on the value set for `max_features` and on the number of features in the problem. Call the number of features in the problem `nFeatures`. Then:

  - If the type of `max_features` is `int`, consider `max_features` features at each split. Note: `max_features gt; nFeatures` throws an error.
  - If the type of `max_features` is `float`, `max_features` is the fraction of features to consider: `int(max_features * nFeatures)`.
  - Possible string values include the following:

    ```
    auto  max_features=nFeatures
    sqrt  max_features=sqrt(nFeatures)
    log2   max_features=log2(nFeatures)
    ```

- If `max_features=None`, then `max_features=nFeatures`.

  Brieman and Cutler[2] recommend `sqrt(nFeatures)` for regression problems. The answers aren't generally terribly sensitive to `max_features`, but this parameter can have some effect, so you'll want to test a few alternative values.

- **random_state**

  int, RandomState instance, or None (default=None)

  - If the type is integer, the integer is used as the seed for the random number generator.
  - If the `random_state` is an instance of `RandomState`, that instance is used as the random number generator.
  - If `random_state` is `None`, the random number generator is the instance of `RandomState` used by `numpy.random`.

`RandomForestRegressor` has several attributes, including the trained trees that make up the ensemble. There's a predict method that will use the trained trees to make predictions, so you will not generally access those directly. You will want to access the variable `importances`. Here is a description:

- **feature_importances**

  This is an array whose length is equal to the number of features in the problem (called `nFeatures` earlier). The values in the array are positive floats indicating relative importance of the corresponding attribute. The `importance`s are determined by a procedure Brieman invented in the original paper on Random Forests.[2] The basic idea is that, one at a time, values of each attribute are randomly permuted, and the change in the model's prediction accuracy is determined. The more the prediction accuracy suffers, the more important the attribute.

Here are descriptions of the methods used:

- **fit(XTrain, yTrain, sample_weight=None)**

  `XTrain` is an array of attribute values. It has `nInstances` rows and `nFeature` columns. `yTrain` is an array of targets. `y` also has `nInstances` rows. In the examples you'll see in this chapter, `yTrain` will have a single column, but the method can fit several models having different targets. For that, `y` would have `nTargets` columns—one column for each set of outcomes. `sample_weight` makes it possible to assign different weights to each of the instances in the training data. It can take one of two forms. The default value of `None` results in equal weighting of all input instances. To apply different weights to each instance, `sample_weight` should be an array with `nInstances` rows and one column.

- **predict(XTest)**

  `XTest` is an array of attribute values for which predictions are produced. The array input to `predict()` has the same number of columns as the array used in `fit()` method for training, but can have a different number of rows, including perhaps a single row. The rows in the output from `predict()` have the same form as rows in the target array `y` used in training.

Modeling Wine Taste with RandomForestRegressor

Listing 7-1 shows how to use the sklearn version of the Random Forest algorithm to build an ensemble model to predict wine taste.

The code reads the wine data set from UCI data repository; does some manipulation to get the attributes, labels, and attribute names into lists; and converts the lists to numpy arrays as required for input to `RandomForestRegressor`. A side benefit of having these input objects in the form of numpy arrays is that it enables the use of a sklearn utility `train_test_split` for building training and test versions of the inputs. The code sets `random_state` to a specified integer value instead of letting the random number generator pick an unrepeatable internal value. That's so that you'll get the same graphs and numeric values when you run the code as the results shown here. Setting `random_state` can also prove handy during development because randomness in the results can mask changes you're making. During real model training, you'll probably want to set `random_state` to its default value, `None`. Fixing `random_state` fixes the holdout set and, as a result, repeated parameter adjustments and retraining may start to overtrain on your holdout set.

The next step in the code is to define a list of ensemble sizes to produce performance graphs that show how the performance varies as the number of trees in the ensemble is changed. For producing detailed plots, the number chosen in Listing 7-2 results in roughly 45 separate runs. That many are useful here so that you can see the shape of the curve of error versus number of trees, but now that you've got that mental picture, you won't want to run so many points in the curve. You might run two or three different numbers of trees early in the development process and then settle on a good number and only run for a single value most of the time.

Most of the parameters affecting training are set as part of the constructor that instantiates a `RandomForestRegressor` object. The call to the constructor is pretty simple in this case. The only parameter that is not left at default values is the `max_features` parameter. The default value (`None`) results in all the features being considered at each node of the tree, which means that it's actually implementing Bagging[3] because no random selection of attributes is involved.[4, 5, 6]

After instantiating a `RandomForestRegressor` object, the next step is to invoke the `fit()` method the training sets as arguments. Once that is done, invoking

the `predict()` method with the attributes from the test set generates predictions that can be compared to the test set labels. The code in the listing uses the sklearn.metrics function `mean_squared_error` to calculate the prediction error. The resulting mean squared error numbers are collected in a list and then plotted. Figure 7-1 shows the resulting plot.

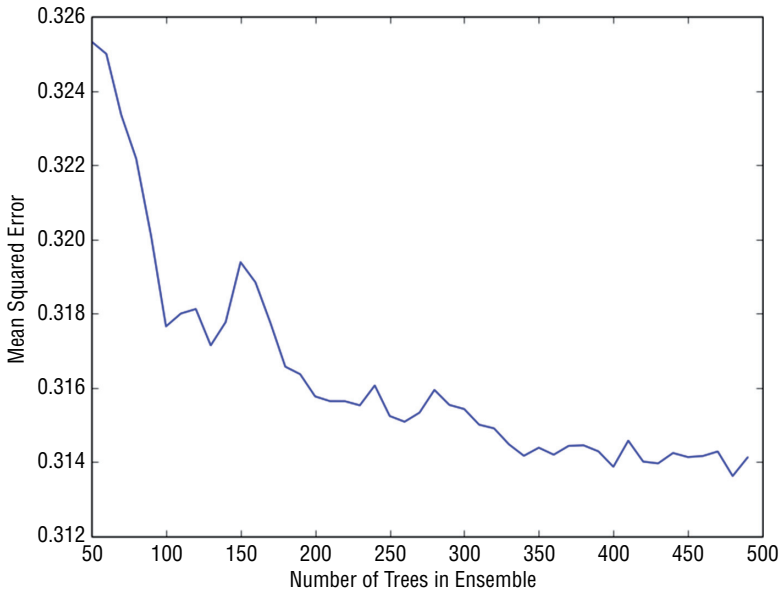

Figure 7-1: Wine taste prediction performance with Random Forest: errors versus ensemble size

The last value of mean squared error is also printed and copied at the bottom of Listing 7-1. Notice that the last value is printed as representative of the mean squared error, not the minimum value. Random Forest generates somewhat independent predictions and then averages them. Adding more trees to the average cannot lead to overfitting, so the minimum point in the curve of Figure 7-1 represents deviation due to statistical fluctuation, not a reproducible minimum.

Listing 7-1: Using RandomForestRegressor to Build a Regression Model—wineRF.py

```
import urllib2
import numpy
from sklearn.cross_validation import train_test_split
from sklearn import ensemble
from sklearn.metrics import mean_squared_error
import pylab as plot


# Read wine quality data from UCI website
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])

X = numpy.array(xList)
y = numpy.array(labels)
wineNames = numpy.array(names)

#take fixed holdout set 30% of data rows
xTrain, xTest, yTrain, yTest = train_test_split(X, y, test_size=0.30,
  random_state=531)

#train Random Forest at a range of ensemble sizes in order to
#see how the mse changes
mseOos = []
nTreeList = range(50, 500, 10)
for iTrees in nTreeList:
    depth = None
    maxFeat  = 4 #try tweaking
    wineRFModel = ensemble.RandomForestRegressor(n_estimators=iTrees,
        max_depth=depth, max_features=maxFeat,
        oob_score=False, random_state=531)

    wineRFModel.fit(xTrain,yTrain)

    #Accumulate mse on test set
    prediction = wineRFModel.predict(xTest)
    mseOos.append(mean_squared_error(yTest, prediction))


print("MSE" )
```

continues

continued

```
print(mseOos[-1])


#plot training and test errors vs number of trees in ensemble
plot.plot(nTreeList, mseOos)
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Mean Squared Error')
#plot.ylim([0.0, 1.1*max(mseOob)])
plot.show()

# Plot feature importance
featureImportance = wineRFModel.feature_importances_

#scale by max importance
featureImportance = featureImportance / featureImportance.max()
sorted_idx = numpy.argsort(featureImportance)
barPos = numpy.arange(sorted_idx.shape[0]) + .5
plot.barh(barPos, featureImportance[sorted_idx], align='center')
plot.yticks(barPos, wineNames[sorted_idx])
plot.xlabel('Variable Importance')
plot.show()

#printed output
#MSE
#0.314125711509
```

Visualizing the Performance of a Random Forests Regression Model

The curve in Figure 7-1 demonstrates the variance reduction properties of the Random Forest algorithm. The level of the error decreases as more trees are added, and the amount of statistical fluctuation in the curve also decreases.

NOTE **To get a feel for the behavior of the algorithm, try changing some of the parameters used in Listing 7-1 and see how the plots change. Try running more trees to see whether you can reduce the error further. Try something like** `nTreeList=range(100,1000,100)`**. Try altering the tree depth parameter to see how sensitive the answers are to tree depth. The wine quality data set has roughly 1,600 instances (rows), so a depth of 10 or 11 could result in almost every point having its own leaf node. A depth of 8 could ideally have 256 leaf nodes, so each one would have an average of about 6 instances. Try some depths in that range to determine whether it affects performance.**

Random Forest generates estimates of how important each variable is to the accuracy of predictions. Listing 7-1 extracts the data member `feature_importance_`, rescales importance values to between `0` and `1`, orders the resulting importance values, and then plots them in a bar chart. Figure 7-2 shows that plot. The most important variable has scaled importance of `1.0` and is the top bar in

the bar chart. It shouldn't be too surprising that alcohol is the most important variable in the Random Forest model. It was also the most important in the penalized linear regression models that you saw in Chapter 5.

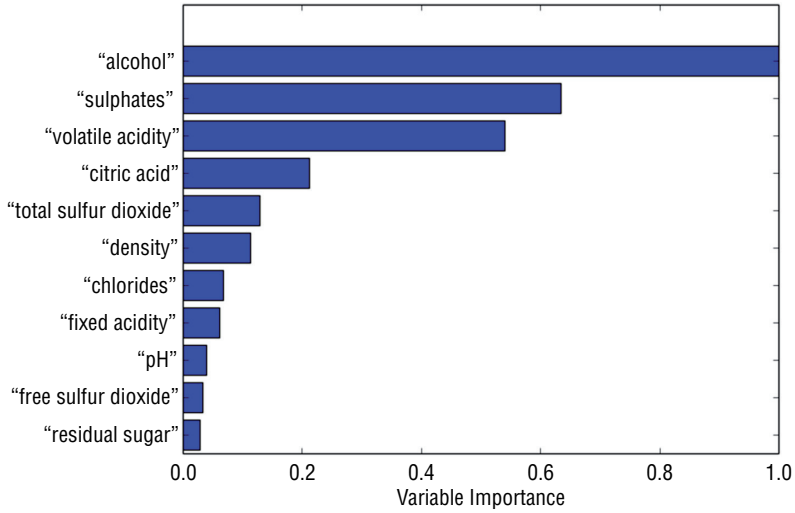

Figure 7-2: Relative importance of variables for Random Forest predicting wine taste

Using Gradient Boosting to Predict Wine Taste

As you saw in Chapter 6, Gradient Boosting[7,8] takes an error-minimization approach to building an ensemble of trees, instead of the variance-reduction approach that Bagging and Random Forest take. Because Gradient Boosting incorporates binary trees as its base learners, it shares some tree-related parameters. However, because Gradient Boosting takes steps directed by the gradient, you'll also see parameters such as step size. In addition, Gradient Boosting's error-minimization approach will lead to different rationale and choices for setting tree depth. There's also a surprise variable that allows you to build models that are a hybrid between Random Forest and Gradient Boosting. You can use Gradient Boosting error-minimization structure while employing the Random Forest random attribute selection for base learners. The sklearn ensemble module is the only place that I've seen that combination available.

Using the Class Constructor for GradientBoostingRegressor[9]

Here is the class constructor for `sklearn.ensemble.GradientBoostingRegressor`:

```
class sklearn.ensemble.GradientBoostingRegressor(loss='ls', learning_
rate=0.1, n_estimators=100, subsample=1.0, min_samples_split=2, min_
samples_leaf=1, max_depth=3, init=None, random_state=None, max_features=
None, alpha=0.9, verbose=0, max_leaf_nodes=None, warm_start=False)
```

The following lists describe the parameters and methods that you'll want to be familiar with and give some comment on the choices and tradeoffs for them where appropriate. This list describes the parameters:

- **loss**

  string, optional (default='ls')

  Gradient Boosting uses trees to approximate the gradient of an overall loss function. The most commonly used overall loss is sum squared error, as is the penalty from ordinary least squares regression. Least sum squared error is a handy choice because the squared error makes the math work out neatly. But other loss functions may better describe your real problem. As an example, I worked on algorithms for automated trading and noticed that using squared error penalty led to algorithms that would avoid large losses but that would accept small losses that in aggregate were more significant. Sum of absolute value of the errors gave much better overall performance; it better matched the real problem. Least mean absolute value is generally less sensitive to outliers. Gradient Boosting is one of the few algorithms that gives you wide flexibility in your choice of penalty functions.

  Possible string values include the following:

  - `ls` Least mean squared error.
  - `lad` Least mean absolute value of error.
  - `huber` Huberized loss is a hybrid between squared error for small values and absolute value of error for large values.[10,11]
  - `quantile` Quantile regression. Predicts quantile (indicated by alpha parameter).

- **learning_rate**

  float, optional (default=0.1)

  As mentioned, Gradient Boosting is based on a gradient descent algorithm. The learning rate is the size of the step taken in the gradient direction. If it is too large, you'll see a rapid decline in the error and then a rapid rise in the error (as a function of the number of trees in the ensemble.) If it is too small, the errors will decrease very slowly, and it will require training more trees than necessary. The best value for `learning_rate` is problem dependent and also depends on the tree depth chosen. The default value of `0.1` is a relatively large value, but a good choice for a starting point. Try it. See whether it leads to instability and overfitting. Adjust if necessary.

- **n_estimators**

  int, optional (default=100)

  This parameter is the number of trees in the ensemble. As you saw in Chapter 6, you can also think of this as the number of steps taken toward

the minimum in a gradient descent sequence. It is also the number of terms in an additive approximation (that is, the sum of the trained models). Because each successive approximation (each successive tree) gets multiplied by the learning rate, a larger learning rate requires fewer trees to be trained to make the same progress toward the minimum. However (as discussed in the section on learning rate), if the learning rate is too high, it can lead to overfitting and may achieve the best performance. It usually takes a few tries to learn what parameter ranges work best on a new problem. The default value of `100` is a good starting point (particularly in conjunction with the default value for the learning rate).

- **subsample**

  float, optional (default=1.0)

  Gradient Boosting becomes stochastic Gradient Boosting when the individual trees are trained on a subsample of the data, similar to Random Forest. Friedman (algorithm inventor) recommends using `subsample=0.5`.[12] That's a good starting point.

- **max_depth**

  integer, optional (default=3)

  As with Random Forests, `max_depth` is the depth of the individual trees in the ensemble. As you saw in the simple example in Chapter 6, Random Forests needs some tree depth to generate a high-fidelity model, whereas Gradient Boosting, by continually focusing on the residual error, was able to get a high-fidelity approximation with trees of depth `1` (called *stumps*). Gradient Boosting's need for deep trees is driven by the degree of interaction between attributes. If they act independently, a depth of 1 will get as good a model as depth 2. Generally, you want to start with a tree depth equal to `1` to get the other parameters set and then try a tree depth of `2` to see whether it gives you an improvement. I've never encountered a problem that needed depth `10`.

- **max_features**

  int, float, string, or None, optional (default = None)

  The number of features to consider when looking for the best split depends on the value set for `max_features` and on the number of features in the problem. Call the number of features in the problem `nFeatures`. Then:

  - If the type of `max_features` is `int`, consider `max_features` at each split.
  - If the type of `max_features` is `float`, then `max_features` is the fraction of features to consider: `int(max_features * nFeatures)`.
  - Possible string values include the following:

    `auto  max_features=nFeatures`

```
sqrt   max_features=sqrt(nFeatures)
log2   max_features=log2(nFeatures)
```

- If `max_features=None`, then `max_features=nFeatures`.

  `max_features` in the Python implementation of Gradient Boosting plays the same role as in Random Forest. It determines how many attributes will be considered for splitting at each node in the trees. That gives the Python implementation of Gradient Boosting a unique capability. It can incorporate Random Forest base learners in the place of trees grown on the full attribute space.

- **warm_start**

  bool, optional(default=False)

  If `warm_start` is set to `True`, subsequent applications of the `fit()` function start from last stopping point in training and continue to accumulate the results of adding further gradient steps.

Here are descriptions of the attributes used:

- **feature_importances**

  An array whose length is equal to the number of features in the problem (called `nFeatures` earlier). The values in the array are positive floats indicating relative importance of the corresponding attribute. A large number corresponds to large influence.

- **train_score**

  An array whose length is equal to the number of trees in the ensemble. This contains the error on the training set at each stage in training the sequence of trees.

Here are descriptions of the methods used:

- **fit(XTrain, yTrain, monitor=None)**

  `XTrain` and `yTrain` have the same form as for Random Forest. `XTrain` is an (`nInstances` x `nAttributes`) numpy array, where `nInstances` is the number of rows in the training data set and `nAttributes` is the number of attributes. `yTrain` is an (`nInstances` x `1`) numpy array of targets. The object "monitor" is a callable that can be used to stop training early.

- **predict(X)**

  `predict(X)` generates a prediction from an array of attributes `X`. `X` needs to have the same number of columns (attributes) as the training set. `X` can have any number of rows.

- **staged_predict(X)**

  This function acts like the `predict()` function except that it's iterable and generates a sequence of predictions corresponding to the sequence of

model produced by the Gradient Boosting algorithm. Each call generates a prediction incorporating one additional tree in the sequence generated by Gradient Boosting.

Getting the parameters set for Gradient Boosting can be a little bewildering for a new user. The following list suggests a sequence of parameter settings and adjustments for Gradient Boosting:

1. Start with default settings, except set `subsample=0.5`. Train a model and look at the curve of out-of-sample (oos) performance versus the number of trees in the ensemble. After the first and subsequent runs, look at the shape of the oos performance curve.
2. If the oos performance is improving rapidly at the right end of the graph either increase `n_estimators` or increase `learning_rate`.
3. If the oos performance is deteriorating rapidly at the right end of the graph, decrease `learning_rate`.
4. Once the oos performance curve improves over its whole length (or only deteriorates very slightly) and levels out at the right side of the graph, try altering `max_depth` and `max_features`.

Using GradientBoostingRegressor to Implement a Regression Model

Listing 7-2 shows what's required to build a Gradient Boosting model for the wine quality data set.

Listing 7-2: Using Gradient Boosting to Build a Regression Model—wineGBM.py

```
import urllib2
import numpy
from sklearn.cross_validation import train_test_split
from sklearn import ensemble
from sklearn.metrics import mean_squared_error
import pylab as plot

# Read wine quality data from UCI website
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(";")
```

continues

continued

```
        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])

X = numpy.array(xList)
y = numpy.array(labels)
wineNames = numpy.array(names)

#take fixed holdout set 30% of data rows
xTrain, xTest, yTrain, yTest = train_test_split(X, y, test_size=0.30,
    random_state=531)

# Train Gradient Boosting model to minimize mean squared error
nEst = 2000
depth = 7
learnRate = 0.01
subSamp = 0.5
wineGBMModel = ensemble.GradientBoostingRegressor(n_estimators=nEst,
                                max_depth=depth,
                                learning_rate=learnRate,
                                subsample = subSamp,
                                loss='ls')

wineGBMModel.fit(xTrain, yTrain)

# compute mse on test set
msError = []
predictions = wineGBMModel.staged_predict(xTest)
for p in predictions:
    msError.append(mean_squared_error(yTest, p))

print("MSE" )
print(min(msError))
print(msError.index(min(msError)))

#plot training and test errors vs number of trees in ensemble
plot.figure()
plot.plot(range(1, nEst + 1), wineGBMModel.train_score_,
    label='Training Set MSE')
```

```
plot.plot(range(1, nEst + 1), msError, label='Test Set MSE')
plot.legend(loc='upper right')
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Mean Squared Error')
plot.show()

# Plot feature importance
featureImportance = wineGBMModel.feature_importances_

# normalize by max importance
featureImportance = featureImportance / featureImportance.max()
idxSorted = numpy.argsort(featureImportance)
barPos = numpy.arange(idxSorted.shape[0]) + .5
plot.barh(barPos, featureImportance[idxSorted], align='center')
plot.yticks(barPos, wineNames[idxSorted])
plot.xlabel('Variable Importance')
plot.subplots_adjust(left=0.2, right=0.9, top=0.9, bottom=0.1)
plot.show()

# Printed Output:
# for:
#nEst = 2000
#depth = 7
#learnRate = 0.01
#subSamp = 0.5
#
# MSE
# 0.313361215728
# 840
```

The first section of code follows the same process as for Random Forest: read the data set, separate the attribute matrix from the targets, convert to numpy arrays, and then form train and test subsets. The training sequence is a little simpler for Gradient Boosting. The code for Random Forest used a loop to generate several models for different values of `n_estimator` to see how the oos error behaved as a function of the number of trees in the ensemble. The Python Gradient Boosting implementation has an iterable (`staged_predict` for regression problems and `staged_decision_function` for classification problems) that simplifies that process. Using these functions, you can train a model incorporating `n_estimator` trees and then generate the oos performance curve for models of all sizes (not greater than `n_estimator`).

Assessing the Performance of a Gradient Boosting Model

Figure 7-3 and the printed output shown in Listing 7-2 show that Gradient Boosting gets about the same level of performance as Random Forest. This is usually the case. There are problems where one or the other will achieve

significantly better performance, so you might want to try them both to make sure. The plot in Figure 7-3 shows an oos error that increases very slightly on the right side of the plot. To see that it is increasing requires looking at the numbers. The increase is not enough to warrant reducing `learning_rate` and retraining.

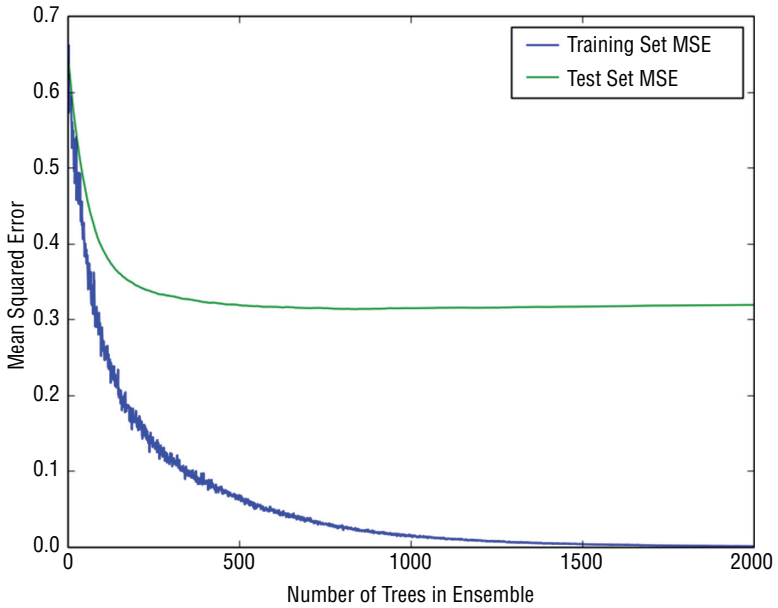

Figure 7-3: Wine taste prediction performance with Gradient Boosting: errors versus ensemble size

Figure 7-4 shows the variable importance determined as part of the Gradient Boosting implementation. Comparing with the variable importance generated by Random Forest reveals that the two are fairly similar but not identical. They agree that the most important variable is alcohol and have several of the same variables in the top four or five.

Coding Bagging to Predict Wine Taste

Listing 7-3 shows the code for generating a bootstrap sample from the wine data, training trees on it and then averaging the resulting models. This is called *Bagging*. It's purely a variance reduction technique, and it's useful to compare the performance that bagging achieves with the performance of Random Forest and Gradient Boosting.

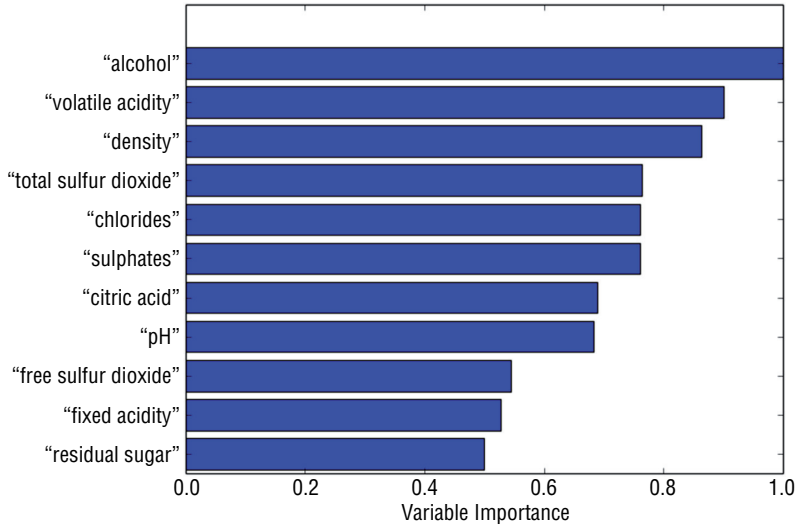

Figure 7-4: Relative importance of variables for Gradient Boosting predicting wine taste

Listing 7-3: Building a Regression Model for Wine Taste Using Bagging (Bootstrap Aggregation)—wineBagging.py

```
__author__ = 'mike-bowles'

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


# Read wine quality data from UCI website
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
```

continues

continued

```
        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 = 100

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

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

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

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))]

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()

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


#With treeDepth = 5
#     bagFract = 0.5
#Minimum MSE
#0.429310223079

#With treeDepth = 8
#     bagFract = 0.5
#Minimum MSE
#0.395838627928

#With treeDepth = 10
#     bagFract = 1.0
#Minimum MSE
#0.313120547589
```

Listing 7-3 includes three parameters that can be tweaked. The first is `numTreesMax`, which determines the number of trees that will be built; the second is `treeDepth`; the third is `bagFract`. As discussed in Chapter 6, Bagging operates by taking a bootstrap sample from the input data. The sample is taken with replacement so some of the data may be repeated. The variable `bagFract` determines how many samples are taken. The original paper on the algorithm recommended that the bootstrap samples be the same size as the original data set, which would correspond to `bagFract = 1.0`. The program in the listing generates `numTreesMax` models from the trees it builds. The first model is the first tree. The second model is the average of the first two trees. The third model is the average of the first three trees, etc. Then the program plots curves of the error versus the number of trees in the model.

Figures 7-5 and 7-6 plot the results for two different parameters settings. The ensemble for Bagging applied to the wine taste prediction data set. The printed results are shown at the bottom of the code listing. Figure 7-5 shows performance versus number of trees where the trees are depth 10 and are trained on bootstrap sample sets as large as the original data set (`bag fraction = 1.0`). With these parameters, Bagging achieves the same level of performance as Random Forest and Gradient Boosting.

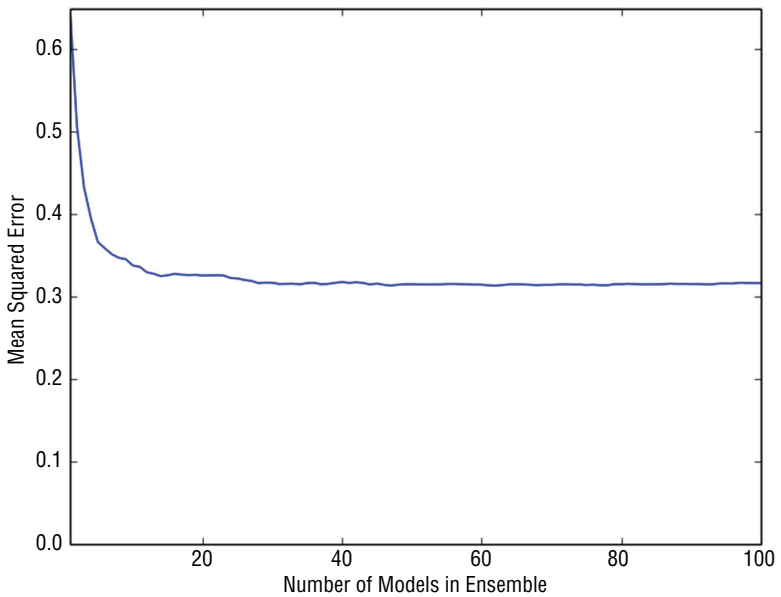

Figure 7-5: Wine taste error for Bagged regression trees (tree depth = 10, bag fraction = 1.0)

Figure 7-6 shows the performance for Bagging using trees of depth 8 and bootstrap data sets half as large as the original data (`bag fraction = 0.5`). As the plot and the printed results indicate, the performance is noticeably worse with this parameter selection.

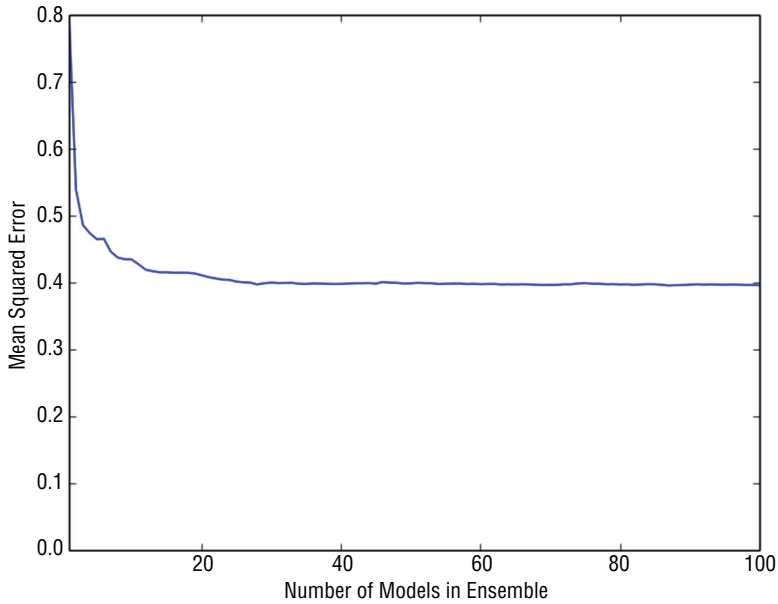

Figure 7-6: Wine taste error for Bagged regression trees (tree depth = 8, bag fraction = 0.5)

Incorporating Non-Numeric Attributes in Python Ensemble Models

Non-numeric attributes are ones that take several discrete non-numeric values. A census record has myriad non-numeric attributes—married, single, divorced, for example; state in which the household is located is another. Non-numeric attributes can improve prediction accuracy, but Python ensemble methods need numeric input. In Chapters 4, "Penalized Linear Regression," and 5, "Building Predictive Models Using Penalized Linear Methods," you saw how to code factor variables so that they could be incorporated in penalized linear regression. The same technique will work here. The problem of estimating the age of abalone will serve as an example to illustrate the technique.

Coding the Sex of Abalone for Input to Random Forest Regression in Python

Suppose that your problem has an attribute that takes n values. The attribute "States in the US" takes 50 values, and "Marital Status" takes 3. To code the n-valued factor variable, you create $n - 1$ new dummy attributes. If the variable takes its ith value, the ith dummy variable is 1 and all other dummies are 0. If the factor variable takes its nth value, all the dummy variables are 0. The abalone data will illustrate.

Listing 7-4 shows the steps training a Random Forest model to predict abalone age from data on the abalone's weight, shell size, and so forth. The objective in this problem is to predict the age of the abalone from various physical measurements (weights of various parts of the abalone, dimensions, and so on). That makes this a regression problem amenable to the algorithms used for building models for predicting taste scores for wines in the previous two sections.

Listing 7-4: Predicting Abalone Age with Random Forest—abaloneRF.py

```
__author__ = 'mike_bowles'

import urllib2
from pylab import *
import matplotlib.pyplot as plot
import numpy
from sklearn.cross_validation import train_test_split
from sklearn import ensemble
from sklearn.metrics import mean_squared_error


target_url = ("http://archive.ics.uci.edu/ml/machine-learning-"
"databases/abalone/abalone.data")
#read abalone data
data = urllib2.urlopen(target_url)

xList = []
labels = []
for line in data:
    #split on semi-colon
    row = line.strip().split(",")

    #put labels in separate array and remove label from row
    labels.append(float(row.pop()))

    #form list of list of attributes (all strings)
    xList.append(row)

#code three-valued sex attribute as numeric
xCoded = []
for row in xList:
    #first code the three-valued sex variable
    codedSex = [0.0, 0.0]
    if row[0] == 'M': codedSex[0] = 1.0
    if row[0] == 'F': codedSex[1] = 1.0

    numRow = [float(row[i]) for i in range(1,len(row))]
    rowCoded = list(codedSex) + numRow
    xCoded.append(rowCoded)
```

```
#list of names for
abaloneNames = numpy.array(['Sex1', 'Sex2', 'Length', 'Diameter',
    'Height', 'Whole weight', 'Shucked weight', 'Viscera weight',
    'Shell weight', 'Rings'])

#number of rows and columns in x matrix
nrows = len(xCoded)
ncols = len(xCoded[1])

#form x and y into numpy arrays and make up column names
X = numpy.array(xCoded)
y = numpy.array(labels)

#break into training and test sets.
xTrain, xTest, yTrain, yTest = train_test_split(X, y, test_size=0.30,
    random_state=531)

#train Random Forest at a range of ensemble sizes in
#order to see how the mse changes
mseOos = []
nTreeList = range(50, 500, 10)
for iTrees in nTreeList:
    depth = None
    maxFeat  = 4 #try tweaking
    abaloneRFModel = ensemble.RandomForestRegressor(n_estimators=iTrees,
        max_depth=depth, max_features=maxFeat,
         oob_score=False, random_state=531)

    abaloneRFModel.fit(xTrain,yTrain)

    #Accumulate mse on test set
    prediction = abaloneRFModel.predict(xTest)
    mseOos.append(mean_squared_error(yTest, prediction))


print("MSE" )
print(mseOos[-1])


#plot training and test errors vs number of trees in ensemble
plot.plot(nTreeList, mseOos)
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Mean Squared Error')
#plot.ylim([0.0, 1.1*max(mseOob)])
plot.show()

# Plot feature importance
featureImportance = abaloneRFModel.feature_importances_

# normalize by max importance
```

continues

continued

```
featureImportance = featureImportance / featureImportance.max()
sortedIdx = numpy.argsort(featureImportance)
barPos = numpy.arange(sortedIdx.shape[0]) + .5
plot.barh(barPos, featureImportance[sortedIdx], align='center')
plot.yticks(barPos, abaloneNames[sortedIdx])
plot.xlabel('Variable Importance')
plot.subplots_adjust(left=0.2, right=0.9, top=0.9, bottom=0.1)
plot.show()

# Printed Output:
# MSE
# 4.30971555911
```

One of the attributes in the data set is the sex of the abalone. There are three possible values for an abalone's gender: male, female, and infant (although the gender of an abalone is indeterminate in infancy). So, the gender attribute is a three-valued factor variable. In the data set, the gender attribute is one of three character variables: `M`, `F`, or `I`. The section of the program that codes this attribute starts with a list filled with two float zeros. If the attribute value is `M`, the first list element is changed to a `1.0`. If the attribute value is `F`, the second list element is changed to a `1.0`. Otherwise, the list is left with two zeros (that is, if the attribute value is `I`). Then the new two-element list replaces the old character variable and the result is used to build a Random Forest model.

Assessing Performance and the Importance of Coded Variables

Figure 7-7 shows how the mean square prediction error decreases as the number of trees in the Random Forest ensemble is changed. The mean squared error in predicting the age of abalone was 4.31. Compare that to the summary statistics that you saw in Chapter 2. The standard deviation of the age (shell rings) was 3.22, meaning that the mean squared variation in the age was 10.37. Therefore, Random Forest is able to predict about 58% of the squared variation in the age of the abalone in the population that was tested.

Figure 7-8 shows the relative variable importance for the Random Forest model. The gender-related variables that were created to deal with the non-numeric gender variable do not turn out to be very important in this model.

Coding the Sex of Abalone for Gradient Boosting Regression in Python

The process of doing the coding for the gender variable is the same for Gradient Boosting as it was for Random Forest. Listing 7-5 contains the code to train a Gradient Boosting model.

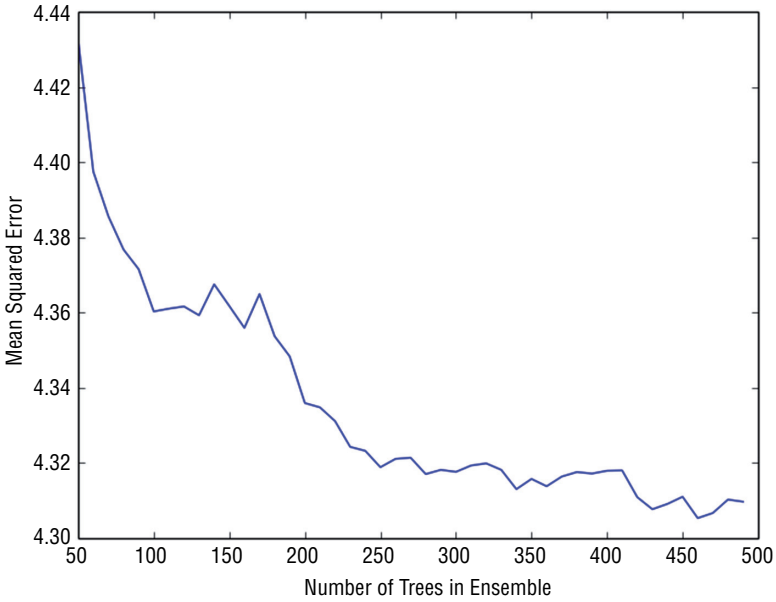

Figure 7-7: Abalone age prediction error with Random Forest

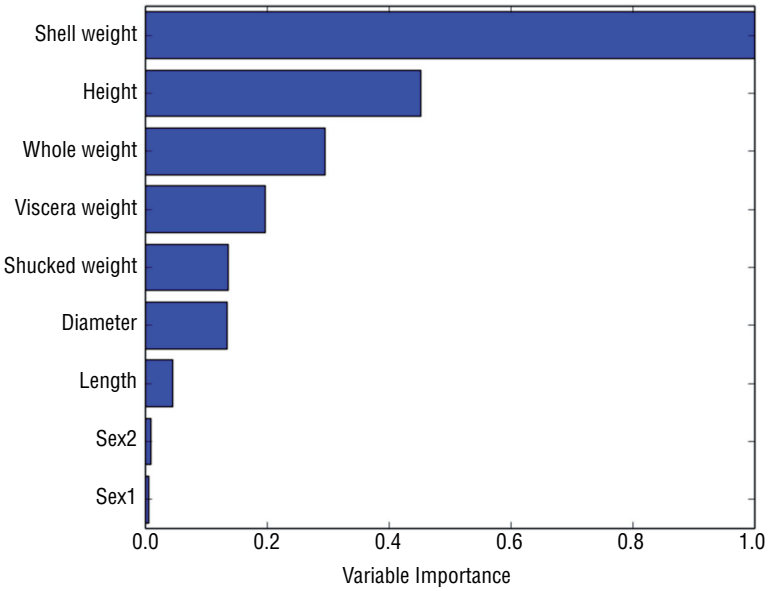

Figure 7-8: Variable importance for abalone age prediction with Random Forest

Listing 7-5: Predicting Abalone Age with Gradient Boosting—abaloneGBM.py

```
__author__ = 'mike_bowles'

import urllib2
```

continues

continued

```
from pylab import *
import matplotlib.pyplot as plot
import numpy
from sklearn.cross_validation import train_test_split
from sklearn import ensemble
from sklearn.metrics import mean_squared_error

target_url = ("http://archive.ics.uci.edu/ml/machine-learning-"
"databases/abalone/abalone.data")
#read abalone data
data = urllib2.urlopen(target_url)

xList = []
labels = []
for line in data:
    #split on semi-colon
    row = line.strip().split(",")

    #put labels in separate array and remove label from row
    labels.append(float(row.pop()))

    #form list of list of attributes (all strings)
    xList.append(row)

#code three-valued sex attribute as numeric
xCoded = []
for row in xList:
    #first code the three-valued sex variable
    codedSex = [0.0, 0.0]
    if row[0] == 'M': codedSex[0] = 1.0
    if row[0] == 'F': codedSex[1] = 1.0

    numRow = [float(row[i]) for i in range(1,len(row))]
    rowCoded = list(codedSex) + numRow
    xCoded.append(rowCoded)

#list of names for
abaloneNames = numpy.array(['Sex1', 'Sex2', 'Length', 'Diameter',
    'Height', 'Whole weight', 'Shucked weight',
    'Viscera weight', 'Shell weight', 'Rings'])

#number of rows and columns in x matrix
nrows = len(xCoded)
ncols = len(xCoded[1])

#form x and y into numpy arrays and make up column names
X = numpy.array(xCoded)
y = numpy.array(labels)

#break into training and test sets.
```

```
xTrain, xTest, yTrain, yTest = train_test_split(X, y, test_size=0.30,
    random_state=531)

#instantiate model
nEst = 2000
depth = 5
learnRate = 0.005
maxFeatures = 3
subsamp = 0.5
abaloneGBMModel = ensemble.GradientBoostingRegressor(n_estimators=nEst,
                        max_depth=depth, learning_rate=learnRate,
                         max_features=maxFeatures,subsample=subsamp,
                         loss='ls')

#train
abaloneGBMModel.fit(xTrain, yTrain)

# compute mse on test set
msError = []
predictions = abaloneGBMModel.staged_decision_function(xTest)
for p in predictions:
    msError.append(mean_squared_error(yTest, p))

print("MSE" )
print(min(msError))
print(msError.index(min(msError)))

#plot training and test errors vs number of trees in ensemble
plot.figure()
plot.plot(range(1, nEst + 1), abaloneGBMModel.train_score_,
    label='Training Set MSE', linestyle=":")
plot.plot(range(1, nEst + 1), msError, label='Test Set MSE')
plot.legend(loc='upper right')
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Mean Squared Error')
plot.show()

# Plot feature importance
featureImportance = abaloneGBMModel.feature_importances_

# normalize by max importance
featureImportance = featureImportance / featureImportance.max()
idxSorted = numpy.argsort(featureImportance)
barPos = numpy.arange(idxSorted.shape[0]) + .5
plot.barh(barPos, featureImportance[idxSorted], align='center')
plot.yticks(barPos, abaloneNames[idxSorted])
plot.xlabel('Variable Importance')
plot.subplots_adjust(left=0.2, right=0.9, top=0.9, bottom=0.1)
plot.show()
```

continues

continued

```
# Printed Output:

# for Gradient Boosting
# nEst = 2000
# depth = 5
# learnRate = 0.003
# maxFeatures = None
# subsamp = 0.5
#
# MSE
# 4.22969363284
# 1736

#for Gradient Boosting with RF base learners
# nEst = 2000
# depth = 5
# learnRate = 0.005
# maxFeatures = 3
# subsamp = 0.5
#
# MSE
# 4.27564515749
# 1687
```

Assessing Performance and the Importance of Coded Variables with Gradient Boosting

There are a couple of things to highlight in the training and results. One is to have a look at the variable importances that Gradient Boosting determines to see whether they agree that the coded gender variables are the least important.

The other thing to check is Python's implementation to incorporate Random Forest base learners for gradient boosting. Will that help or hurt performance? The only thing required to make Gradient Boosting use Random Forest base learners is to change the `max_features` variable from `None` to an integer value less than the number of attributes or a float less than `1.0`. When `max_features` is set to `None`, all nine of the features are considered when the Tree Growing algorithm is searching for the best attribute for splitting the data at each of the nodes. When `max_features` is set to an integer less than 9, the features are chosen from a set attributes of length `max_features` chosen at random for each node.

The printed output from the code in Listing 7-5 is shown at the bottom of the listing. The mean squared error numbers indicate that there's not much performance difference between Random Forest and Gradient Boosting for predicting abalone age. There's also not much difference between using simple trees as base learners versus using Random Forest base learners in Gradient Boosting when they're used to predict abalone age.

The use of simple trees versus Random Forest similarly makes little difference in trajectories of prediction error versus ensemble size, as seen by comparing Figures 7-9 through 7-11.

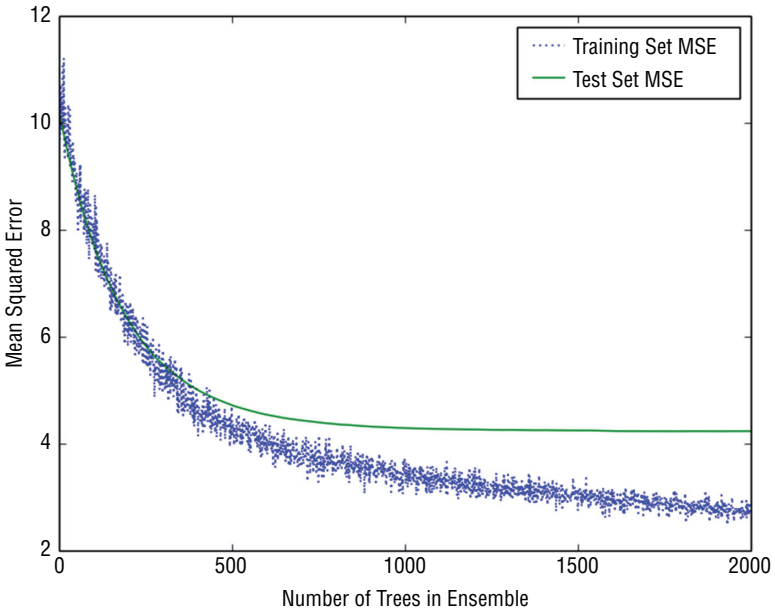

Figure 7-9: Abalone age prediction error with Gradient

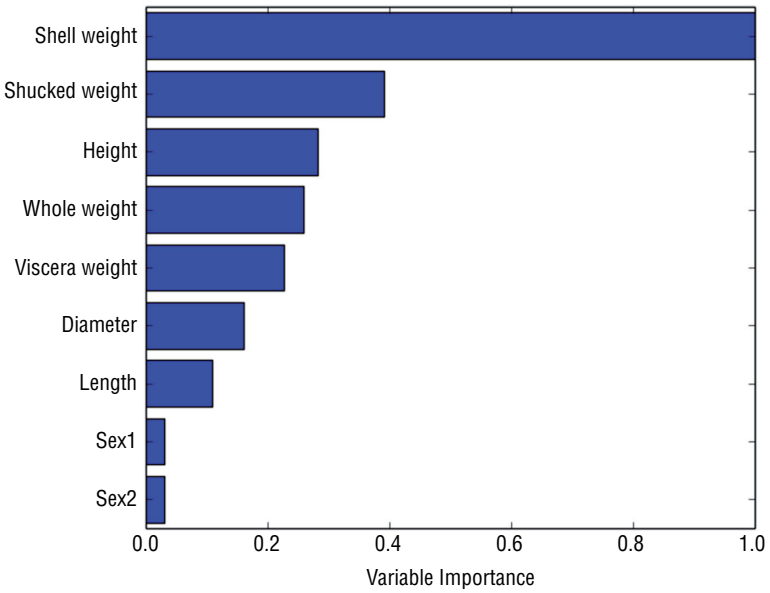

Figure 7-10: Variable importance for abalone age prediction with Gradient Boosting

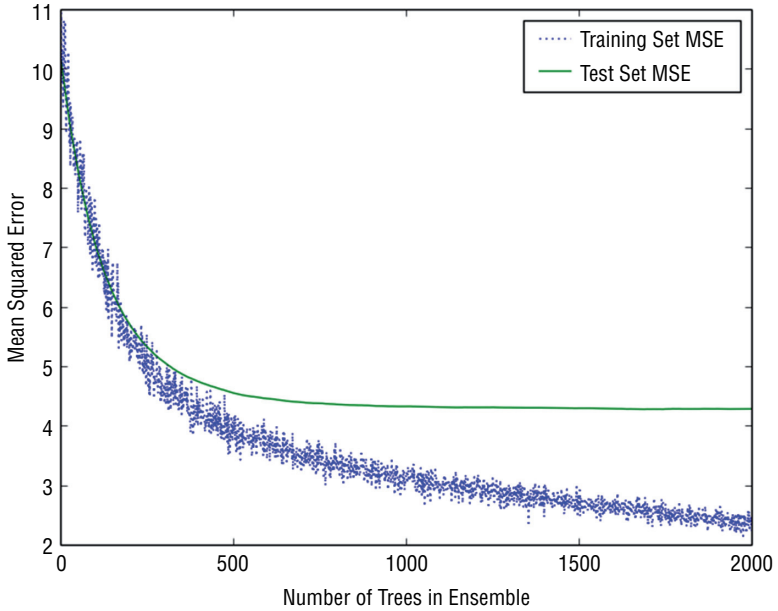

Figure 7-11: Abalone age prediction error with Gradient Boosting using Random Forest base learners

Figures 7-10 and 7-12 show the variable importance for Gradient Boosting based on simple trees and based on Random Forest base learners, respectively. The only difference between the two lists is that viscera weight and height (fourth and fifth most important variables) are swapped in position between the two. Similarly, there is little difference between the order of the importance list generated by Random Forest and either of the two lists generated by Gradient Boosting.

There's some belief that Random Forest has an advantage on wider attribute spaces, particularly for sparse ones such as in text-mining problems. The next section compares the two algorithms on a binary classification problem: classifying rock versus mines using sonar output. That problem has 60 attributes—not as wide as a text-mining problem, but perhaps that will show some performance difference between Gradient Boosting using ordinary binary decision trees versus using Random Forest base learners.

Solving Binary Classification Problems with Python Ensemble Methods

This section covers two basic types of classification problems: binary classification and multiclass classification. Binary classification problems are ones where there are two possible outcomes. Those outcomes might be "clicked on the ad"

or "didn't click on the ad," for example. The example used here to illustrate the use of ensemble methods is the rocks versus mines problem, where the task is to use sonar returns to determine whether the object being scanned by the sonar is a rock or a mine.

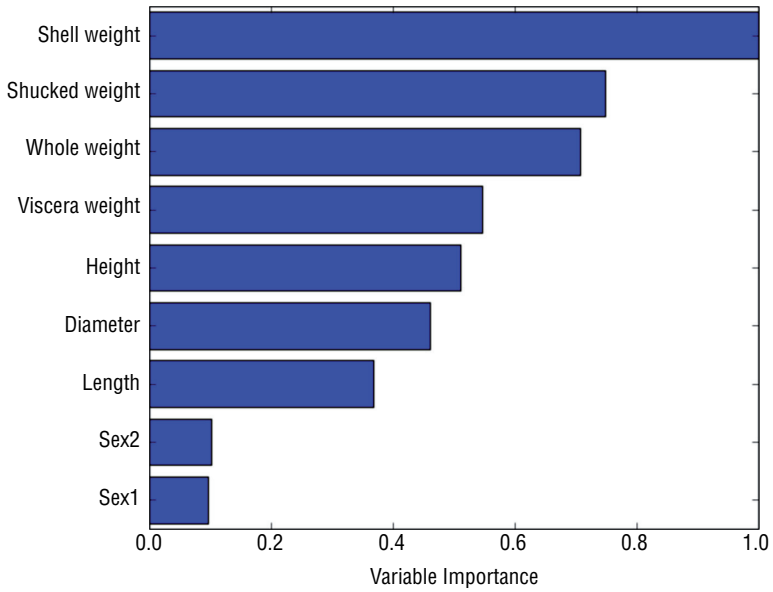

Figure 7-12: Variable importance for abalone age prediction with Gradient Boosting using Random Forest base learners

Multiclass problems are ones where there are more than two possible outcomes. Classifying glass samples according their chemical composition serves to illustrate the use of Python ensemble methods for this class of problems.

Detecting Unexploded Mines with Python Random Forest

The lists that follow show the constructor and its arguments for `RandomForestClassifier`.[13] Most of the arguments for the `RandomForestClassifier` are the same as for `RandomForestRegressor`. The arguments for `RandomForestRegressor` were outlined and discussed in the section on using `RandomForestRegressor` for predicting wine quality. This section highlights only the elements of the `RandomForestClassifier` class that differ from their regression counterparts.

The first difference is the criterion used for judging the quality of splits. Recall from Chapter 6 that the process of training a tree involves trying all possible attributes and all possible split points for each attribute and then picking the

attribute and split point that give the best split. For regression trees, the quality of the split was judged on the basis of sum squared error. Sum squared error does not work for classification problems. Something more like misclassification error is required.

Here is the class constructor for `sklearn.ensemble.RandomForestClassifier`:

```
sklearn.ensemble.RandomForestClassifier(n_estimators=10, criterion=
'gini', max_depth=None, min_samples_split=2, min_samples_leaf=1,
max_features='auto', max_leaf_nodes=None, bootstrap=True, oob_score=
False, n_jobs=1, random_state=None, verbose=0, min_density=None,
compute_importances=None)
```

The following list describes the parameter:

- **criterion**

  string, optional (default='gini')

  Possible values include the following:

  `gini` Use gini impurity measure

  `entropy` Use entropy-based information gain

  For more information on these two measures of node impurity, see the Wikipedia page on binary decision trees at `http://en.wikipedia.org/wiki/Decision_tree_learning`. As a practical matter, the choice does not make a lot of difference for ensemble performance.

Classification trees naturally produce probabilities of class membership based on the percentages of different classes from the training data that wind up in each of the leaf nodes. Depending on the application you have, for the answers you might prefer to work directly with those probabilities or you may want to have the value of the most numerous class returned as the prediction for those examples that wind up in the leaf node. If you're going to adjust thresholds used in conjunction with the prediction, you'll want to have the probabilities. For generating area under the curve (AUC), you'll get better fidelity on the receiver operating curve (ROC) with probabilities. If you want to calculate misclassification errors, you'll want the probabilities converted to a prediction of a specific class.

The following list describes the methods:

- **fit**(X, y, sample_weight=None)

  The description of the arguments for the classification version of Random Forest differs only in the nature of the labels `y`. For a classification problem, the labels are integers taking values from `0` to the number of different classes minus `1`. For binary classification the labels are `0` or `1`. For a multiclass problem with `nClass` different classes they are integers from `0` to `nClass - 1`.

- **predict**(X)

  For an attribute matrix (two-dimensional numpy array) `X`, this function produces a specific class prediction. It yields a single column array with the same number of rows as `X`. Each entry is a predicted class, whether the problem is a binary classification problem or a multiclass problem.

- **predict_proba**(X)

  This version of the prediction function produces a two-dimensional array. The number of rows matches the number of rows in `X`. The number of columns is equal to the number of classes being predicted (two columns for a binary classification problem, for example). The entry in each row is the probability of the associated class.

- **predict_log_proba**(X)

  This version of the prediction function produces a two-dimensional array similar to the `predict_proba`. Instead of showing probabilities, this version shows log of probability.

Constructing a Random Forests Model to Detect Unexploded Mines

Listing 7-8 shows how to build a Random Forest model for detecting unexploded mines using sonar. The overall structure of the data setup and training should be familiar from the other Random Forest examples earlier in this chapter and in Chapter 6. Differences stem from properties of classification problems. First you'll notice that the labels are changed from `M` and `R` to `0` and `1`. That's an input requirement for `RandomForestClassifier`. The next differences show up after training when evaluating performance on the test set. For a binary classification problem, there is choice of using area under the ROC curve (AUC) or misclassification error. I usually prefer AUC when it is available because it gives an overall measure of performance.

To calculate AUC, the `predict_proba` version of the `predict()` function is used. You cannot get a useful ROC curve with predictions that are already reduced to a specific class. (More correctly, the ROC curve you calculate only has three points on it: the two end points and one point in the middle.) The sklearn metric utilities make calculating the AUC simple, with just a couple of lines of code. Those get accumulated into a list to plot AUC performance as a function of the number of trees in the ensemble. The code in Listing 7-7 then plots the AUC versus number of trees, the feature importance for the 30 most important features, and the ROC curve for the largest ensemble of the ones that are generated. The last section of the code picks three different threshold levels and prints out the confusion matrix for each of these threshold levels. The threshold levels are chosen at the three quartile boundaries, and the results

show how false positives and false negatives change as the threshold moves to favor one versus the others.

Listing 7-7: Classifying Sonar Returns as Rocks or Mines with Random Forest—rocksVMinesRF.py

```
__author__ = 'mike_bowles'

import urllib2
from math import sqrt, fabs, exp
import matplotlib.pyplot as plot
from sklearn.cross_validation import train_test_split
from sklearn import ensemble
from sklearn.metrics import roc_auc_score, roc_curve
import numpy

#read data from uci data repository
target_url = ("https://archive.ics.uci.edu/ml/machine-learning"
"databases/undocumented/connectionist-bench/sonar/sonar.all-data")
data = urllib2.urlopen(target_url)

#arrange data into list for labels and list of lists for attributes
xList = []

for line in data:
    #split on comma
    row = line.strip().split(",")
    xList.append(row)

#separate labels from attributes, convert from attributes from
#string to numeric and convert "M" to 1 and "R" to 0

xNum = []
labels = []

for row in xList:
    lastCol = row.pop()
    if lastCol == "M":
        labels.append(1)
    else:
        labels.append(0)
    attrRow = [float(elt) for elt in row]
    xNum.append(attrRow)

#number of rows and columns in x matrix
nrows = len(xNum)
ncols = len(xNum[1])

#form x and y into numpy arrays and make up column names
X = numpy.array(xNum)
y = numpy.array(labels)
```

```
rocksVMinesNames = numpy.array(['V' + str(i) for i in range(ncols)])

#break into training and test sets.
xTrain, xTest, yTrain, yTest = train_test_split(X, y, test_size=0.30,
    random_state=531)

auc = []
nTreeList = range(50, 2000, 50)
for iTrees in nTreeList:
    depth = None
    maxFeat  = 8 #try tweaking
    rocksVMinesRFModel = ensemble.RandomForestClassifier(n_estimators=
                            iTrees, max_depth=depth, max_features=
                            maxFeat, oob_score=False, random_state=531)
                            rocksVMinesRFModel.fit(xTrain,yTrain)

    #Accumulate auc on test set
    prediction = rocksVMinesRFModel.predict_proba(xTest)
    aucCalc = roc_auc_score(yTest, prediction[:,1:2])
    auc.append(aucCalc)

print("AUC" )
print(auc[-1])


#plot training and test errors vs number of trees in ensemble
plot.plot(nTreeList, auc)
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Area Under ROC Curve - AUC')
#plot.ylim([0.0, 1.1*max(mseOob)])
plot.show()

# Plot feature importance
featureImportance = rocksVMinesRFModel.feature_importances_

# normalize by max importance
featureImportance = featureImportance / featureImportance.max()

#plot importance of top 30
idxSorted = numpy.argsort(featureImportance)[30:60]
idxTemp = numpy.argsort(featureImportance)[::-1]
print(idxTemp)
barPos = numpy.arange(idxSorted.shape[0]) + .5
plot.barh(barPos, featureImportance[idxSorted], align='center')
plot.yticks(barPos, rocksVMinesNames[idxSorted])
plot.xlabel('Variable Importance')
plot.show()

#plot best version of ROC curve
fpr, tpr, thresh = roc_curve(yTest, list(prediction[:,1:2]))
```

continues

continued

```
ctClass = [i*0.01 for i in range(101)]

plot.plot(fpr, tpr, linewidth=2)
plot.plot(ctClass, ctClass, linestyle=':')
plot.xlabel('False Positive Rate')
plot.ylabel('True Positive Rate')
plot.show()

#pick some threshold values and calc confusion matrix for
#best predictions

#notice that GBM predictions don't fall in range of (0, 1)
#pick threshold values at 25th, 50th and 75th percentiles
idx25 = int(len(thresh) * 0.25)
idx50 = int(len(thresh) * 0.50)
idx75 = int(len(thresh) * 0.75)

#calculate total points, total positives and total negatives
totalPts = len(yTest)
P = sum(yTest)
N = totalPts - P

print('')
print('Confusion Matrices for Different Threshold Values')

#25th
TP = tpr[idx25] * P; FN = P - TP; FP = fpr[idx25] * N; TN = N - FP
print('')
print('Threshold Value =   ', thresh[idx25])
print('TP = ', TP/totalPts, 'FP = ', FP/totalPts)
print('FN = ', FN/totalPts, 'TN = ', TN/totalPts)

#50th
TP = tpr[idx50] * P; FN = P - TP; FP = fpr[idx50] * N; TN = N - FP
print('')
print('Threshold Value =   ', thresh[idx50])
print('TP = ', TP/totalPts, 'FP = ', FP/totalPts)
print('FN = ', FN/totalPts, 'TN = ', TN/totalPts)

#75th
TP = tpr[idx75] * P; FN = P - TP; FP = fpr[idx75] * N; TN = N - FP
print('')
print('Threshold Value =   ', thresh[idx75])
print('TP = ', TP/totalPts, 'FP = ', FP/totalPts)
print('FN = ', FN/totalPts, 'TN = ', TN/totalPts)


# Printed Output:
#
# AUC
```

```
# 0.950304259635
#
# Confusion Matrices for Different Threshold Values
#
# ('Threshold Value =   ', 0.76051282051282054)
# ('TP = ', 0.25396825396825395, 'FP = ', 0.0)
# ('FN = ', 0.2857142857142857, 'TN = ', 0.46031746031746029)
#
# ('Threshold Value =   ', 0.62461538461538457)
# ('TP = ', 0.46031746031746029, 'FP = ', 0.047619047619047616)
# ('FN = ', 0.079365079365079361, 'TN = ', 0.41269841269841268)
#
# ('Threshold Value =   ', 0.46564102564102566)
# ('TP = ', 0.53968253968253965, 'FP = ', 0.22222222222222221)
# ('FN = ', 0.0, 'TN = ', 0.23809523809523808)
```

Determining the Performance of a Random Forests Classifier

Figure 7-13 shows a plot of AUC versus number of trees. The plot appears upside down from the plots you've seen involving mean squared error or misclassification error. For mean squared error and misclassification error, smaller is better. For AUC, `1.0` is perfect, and `0.5` is perfectly bad. So, with AUC, larger is better, and instead of looking for a valley in the plot, you're looking for a peak. Figure 7-13 shows a peak toward the left side of the plot. However, because Random Forest only reduces variance and does not overfit, the peak can be attributed to random fluctuation. As was the case with some of the regression problem earlier in the chapter, the best choice of model is the one including all the trees whose performance is the rightmost point on the curve.

Figure 7-14 plots the variable importance for the most important 30 variables in the Random Forest mine detector. The different attributes in the mine detection problem correspond to different frequencies of sonar signal and therefore different wavelengths. If you were given the problem of designing the machine learning for this problem, your next step might be to determine the wavelengths corresponding to these variables and compare those wavelengths to the characteristic dimensions of the rocks and mines in the test and training set. That could help you get some faith and understanding of the model.

The model is getting remarkably high AUC, and the ROC curve is correspondingly good. It doesn't quite square the corner in the upper left, but it comes pretty close.

Detecting Unexploded Mines with Python Gradient Boosting

Listing 7-7 shows the form of the constructor for Gradient Boosting in sci-kit learn. Most of the arguments and methods for `GradientBoostingClassifier`[14] are the same as for `GradientBoostingRegressor`, so the following descriptions

are limited to the elements that are different with the classifier than with the regression version.

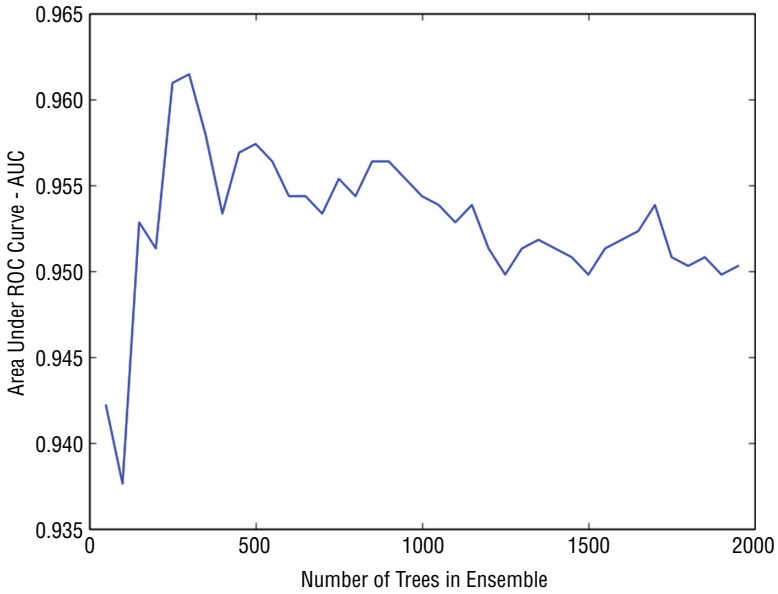

Figure 7-13: AUC versus ensemble size for Random Forest models for detecting mines using sonar

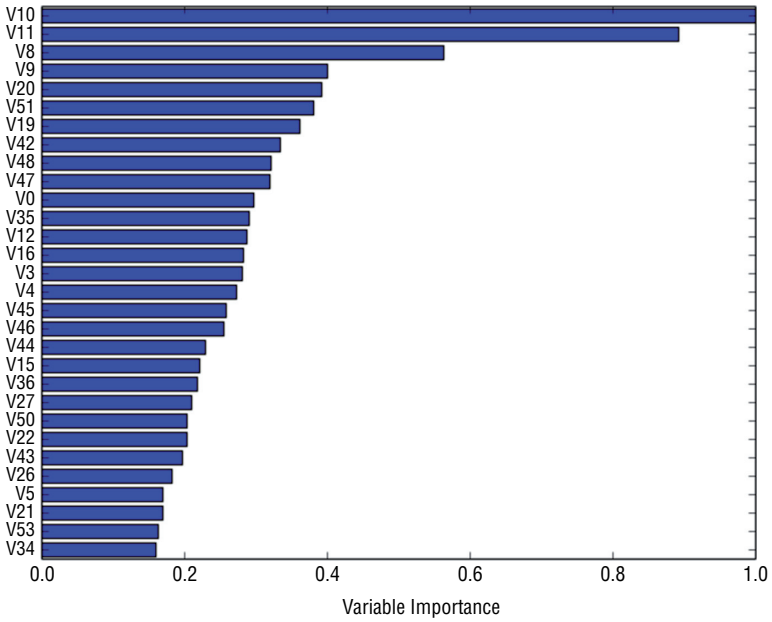

Figure 7-14: Variable importance for Random Forest mine detection model

Here is the class constructor for `sklearn.ensemble.GradientBoostingClassifier`:

```
sklearn.ensemble.GradientBoostingClassifier(loss='deviance', learning_
rate=0.1, n_estimators=100, subsample=1.0, min_samples_split=2,
min_samples_leaf=1, max_depth=3, init=None, random_state=None,
max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False)
```

The following list describes the parameters:

- **loss**

  `deviance` is the default and the only option for classification.

The following list describes the methods:

- **fit**(X, y, monitor=None)

  The description of the arguments for the classification version of Random Forest differs only in the nature of the labels `y`. For a classification problem the labels are integers taking values from `0` to the number of different classes minus `1`. For binary classification the labels are `0` or `1`. For a multiclass problem with `nClass` different classes they are integers from `0` to `nClass - 1`.

- **decision_function**(X)

  Under the hood of a Gradient Boosting classifier is a sum of regression trees. These generate a real number estimate related to the probability of class membership. These real number estimates have to be passed through an inverse logistic function to turn them into probabilities. The real number values before being converted are available through the decision function and can be used just as easily as probabilities for ROC curve calculations.

- **predict**(X)

  This function predicts class membership.

- **predict_proba**(X)

  This function predicts class probabilities. It has a column of probabilities for each class. For a binary problem, there are two columns. For multiclass problems, there are `nClass` columns.

The staged versions of these functions are iterable and will generate as many values as there are trees in the ensemble (which is the same as the number of steps in the training).

- **staged_decision_function**(X)

  This is the staged version of the `decision` function.

- **staged_predict**(X)

  This is the staged version of the `predict` function.

- **staged_predict_proba**(X)

  This is the staged version of the `predict_proba` function.

The code in Listing 7-8 applies the sklearn `GradientBoostingClassifier` to the task of detecting mines.

Listing 7-8: Classifying Sonar Returns as Rocks or Mines with Gradient Boosting—rocksVMinesGBM.py

```
__author__ = 'mike_bowles'

import urllib2
from math import sqrt, fabs, exp
import matplotlib.pyplot as plot
from sklearn.cross_validation import train_test_split
from sklearn import ensemble
from sklearn.metrics import roc_auc_score, roc_curve
import numpy

#read data from uci data repository
target_url = ("https://archive.ics.uci.edu/ml/machine-learning-"
"databases/undocumented/connectionist-bench/sonar/sonar.all-data")
data = urllib2.urlopen(target_url)


#arrange data into list for labels and list of lists for attributes
xList = []


for line in data:
    #split on comma
    row = line.strip().split(",")
    xList.append(row)

#separate labels from attributes, convert from attributes from
#string to numeric and convert "M" to 1 and "R" to 0

xNum = []
labels = []

for row in xList:
    lastCol = row.pop()
    if lastCol == "M":
        labels.append(1)
    else:
        labels.append(0)
    attrRow = [float(elt) for elt in row]
    xNum.append(attrRow)

#number of rows and columns in x matrix
nrows = len(xNum)
ncols = len(xNum[1])

#form x and y into numpy arrays and make up column names
```

```
X = numpy.array(xNum)
y = numpy.array(labels)
rockVMinesNames = numpy.array(['V' + str(i) for i in range(ncols)])

#break into training and test sets.
xTrain, xTest, yTrain, yTest = train_test_split(X, y, test_size=0.30,
    random_state=531)

#instantiate model
nEst = 2000
depth = 3
learnRate = 0.007
maxFeatures = 20
rockVMinesGBMModel = ensemble.GradientBoostingClassifier(
                        n_estimators=nEst, max_depth=depth,
                        learning_rate=learnRate,
                        max_features=maxFeatures)
#train
rockVMinesGBMModel.fit(xTrain, yTrain)

# compute auc on test set as function of ensemble size
auc = []
aucBest = 0.0
predictions = rockVMinesGBMModel.staged_decision_function(xTest)
for p in predictions:
    aucCalc = roc_auc_score(yTest, p)
    auc.append(aucCalc)

    #capture best predictions
    if aucCalc > aucBest:
        aucBest = aucCalc
        pBest = p

idxBest = auc.index(max(auc))

#print best values
print("Best AUC" )
print(auc[idxBest])
print("Number of Trees for Best AUC")
print(idxBest)

#plot training deviance and test auc's vs number of trees in ensemble
plot.figure()
plot.plot(range(1, nEst + 1), rockVMinesGBMModel.train_score_,
    label='Training Set Deviance', linestyle=":")
plot.plot(range(1, nEst + 1), auc, label='Test Set AUC')
plot.legend(loc='upper right')
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Deviance / AUC')
plot.show()
```

continues

continued

```
# Plot feature importance
featureImportance = rockVMinesGBMModel.feature_importances_

# normalize by max importance
featureImportance = featureImportance / featureImportance.max()

#plot importance of top 30
idxSorted = numpy.argsort(featureImportance)[30:60]

barPos = numpy.arange(idxSorted.shape[0]) + .5
plot.barh(barPos, featureImportance[idxSorted], align='center')
plot.yticks(barPos, rockVMinesNames[idxSorted])
plot.xlabel('Variable Importance')
plot.show()

#pick threshold values and calc confusion matrix for best predictions
#notice that GBM predictions don't fall in range of (0, 1)

#plot best version of ROC curve
fpr, tpr, thresh = roc_curve(yTest, list(pBest))
ctClass = [i*0.01 for i in range(101)]

plot.plot(fpr, tpr, linewidth=2)
plot.plot(ctClass, ctClass, linestyle=':')
plot.xlabel('False Positive Rate')
plot.ylabel('True Positive Rate')
plot.show()

#pick threshold values and calc confusion matrix for best predictions
#notice that GBM predictions don't fall in range of (0, 1)
#pick threshold values at 25th, 50th and 75th percentiles
idx25 = int(len(thresh) * 0.25)
idx50 = int(len(thresh) * 0.50)
idx75 = int(len(thresh) * 0.75)

#calculate total points, total positives and total negatives
totalPts = len(yTest)
P = sum(yTest)
N = totalPts - P

print('')
print('Confusion Matrices for Different Threshold Values')

#25th
TP = tpr[idx25] * P; FN = P - TP; FP = fpr[idx25] * N; TN = N - FP
print('')
print('Threshold Value =   ', thresh[idx25])
print('TP = ', TP/totalPts, 'FP = ', FP/totalPts)
print('FN = ', FN/totalPts, 'TN = ', TN/totalPts)

#50th
```

```
TP = tpr[idx50] * P; FN = P - TP; FP = fpr[idx50] * N; TN = N - FP
print('')
print('Threshold Value =   ', thresh[idx50])
print('TP = ', TP/totalPts, 'FP = ', FP/totalPts)
print('FN = ', FN/totalPts, 'TN = ', TN/totalPts)

#75th
TP = tpr[idx75] * P; FN = P - TP; FP = fpr[idx75] * N; TN = N - FP
print('')
print('Threshold Value =   ', thresh[idx75])
print('TP = ', TP/totalPts, 'FP = ', FP/totalPts)
print('FN = ', FN/totalPts, 'TN = ', TN/totalPts)


# Printed Output:
#
# Best AUC
# 0.936105476673
# Number of Trees for Best AUC
# 1989
#
# Confusion Matrices for Different Threshold Values
#
# ('Threshold Value =   ', 6.2941249291909935)
# ('TP = ', 0.23809523809523808, 'FP = ', 0.015873015873015872)
# ('FN = ', 0.30158730158730157, 'TN = ', 0.44444444444444442)
#
# ('Threshold Value =   ', 2.2710265370949441)
# ('TP = ', 0.44444444444444442, 'FP = ', 0.063492063492063489)
# ('FN = ', 0.095238095238095233, 'TN = ', 0.3968253968253968)
#
# ('Threshold Value =   ', -3.0947902666953317)
# ('TP = ', 0.53968253968253965, 'FP = ', 0.22222222222222221)
# ('FN = ', 0.0, 'TN = ', 0.23809523809523808)
#
#
# Printed Output with max_features = 20 (Random Forest base learners):
#
# Best AUC
# 0.956389452333
# Number of Trees for Best AUC
# 1426
#
# Confusion Matrices for Different Threshold Values
#
# ('Threshold Value =   ', 5.8332200248698536)
# ('TP = ', 0.23809523809523808, 'FP = ', 0.015873015873015872)
# ('FN = ', 0.30158730158730157, 'TN = ', 0.44444444444444442)
#
# ('Threshold Value =   ', 2.0281780133610567)
# ('TP = ', 0.47619047619047616, 'FP = ', 0.031746031746031744)
```

continues

continued

```
# ('FN = ', 0.063492063492063489, 'TN = ', 0.42857142857142855)
#
# ('Threshold Value =   ', -1.2965629080181333)
# ('TP = ', 0.53968253968253965, 'FP = ', 0.22222222222222221)
# ('FN = ', 0.0, 'TN = ', 0.23809523809523808)
```

The code follows the same general progression as was followed for Random Forest. One difference is that Gradient Boosting can overfit, and so the program keeps track of the best value of AUC as it accumulates AUCs into a list to be plotted. The best version is then used for generating a ROC curve and the tables of false positives, false negatives, and so on. Another difference is that Gradient Boosting is run twice—once incorporating ordinary trees and once using Random Forest base learners. Both ways have very good classification performance. The version using Random Forest base learners achieved better performance, unlike the models for predicting abalone age, where the performance was not markedly changed.

Determining the Performance of a Gradient Boosting Classifier

Figure 7-16 plots two curves. One is the deviance on the training set. Deviance is related to how far the probability estimates are from correct but differs slightly from misclassification error. Deviance is plotted because that quantity is what gradient boosting is training to improve. It's included in the plot to show the progress of training. The AUC (on oos data) is also plotted to show how the oos performance is changing as the number of trees increases (or equivalently more gradient steps are taken; each step results in training an additional tree).

Figure 7-17 plots the variable importance for the most important 30 variables in the Gradient Boosting mine detector. The variable importances in Figure 7-17 have a somewhat different order than the ones for Random Forest (shown in Figure 7-14). There is some commonality; for example, variables V10, V11, V20 and V51 are near the top of both lists although not in quite the same order.

Figure 7-19 shows model training progress for Gradient Boosting that is using Random Forest base learners. Gradient Boosting does get better results using Random Forest base learners, but the difference isn't large enough to be obvious in the graph.

Using Random Forest base learners doesn't change the variable importance very much, as you can see by comparing Figure 7-20 with Figure 7-17.

Figure 7-21 shows the ROC curve for the mine detector model built with Gradient Boosting using Random Forest base learners.

In this section you have seen how ensemble methods can be used to solve binary classification problems. In most respects, using the application of ensemble methods to binary classification problems is the same as for regression problems. As an illustration of the similarity, notice how many of the parameters required to instantiate a randomForestRegressor object are the same as the ones for a

randomForestClassification object. Based on what you saw in Chapter 6, you can understand the basis for this similarity.

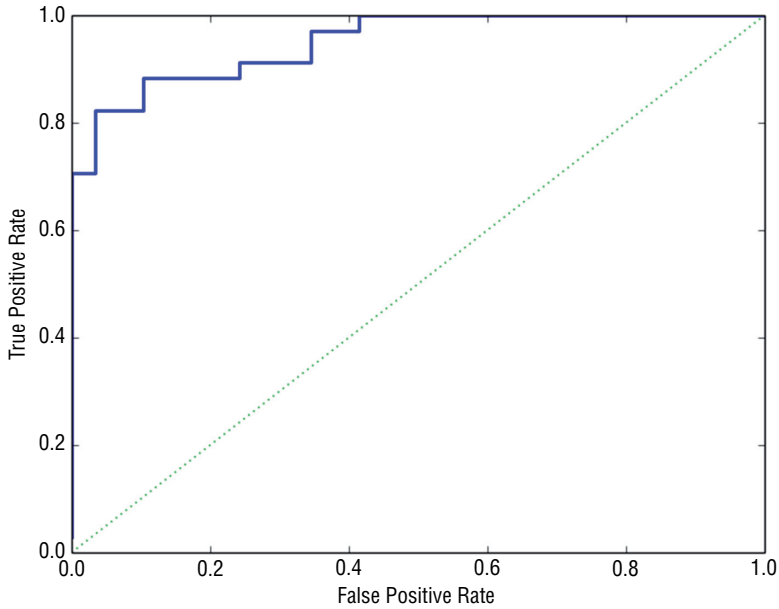

Figure 7-15: ROC curve for Random Forest mine detection model

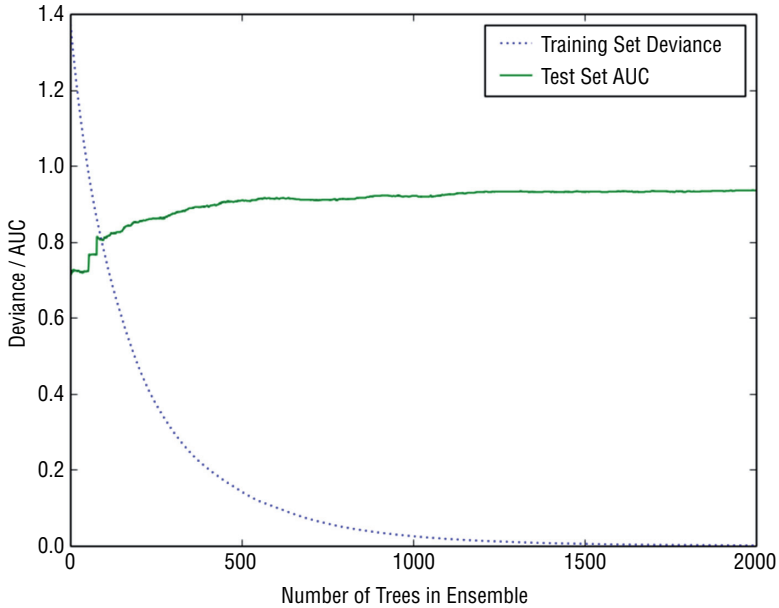

Figure 7-16: AUC versus ensemble size for Gradient Boosting models for detecting mines using sonar

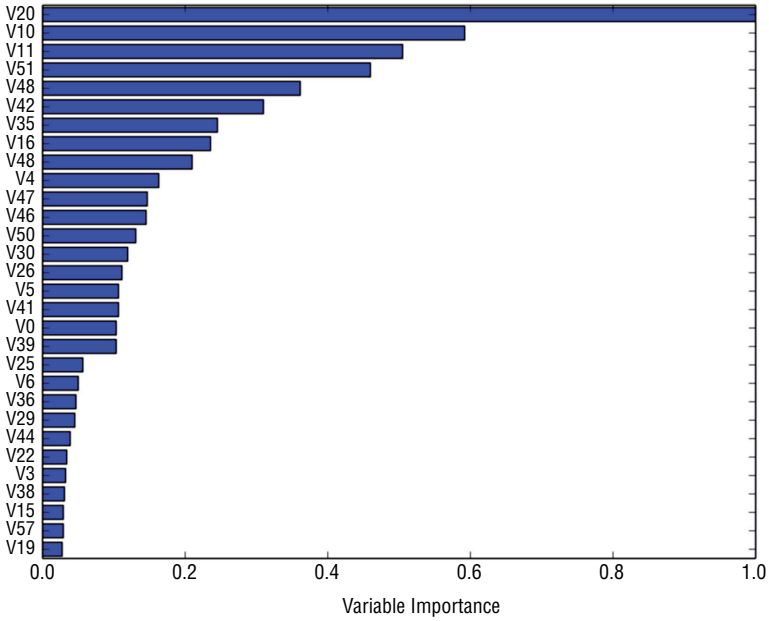

Figure 7-17: Variable importance for Gradient Boosting mine detection model

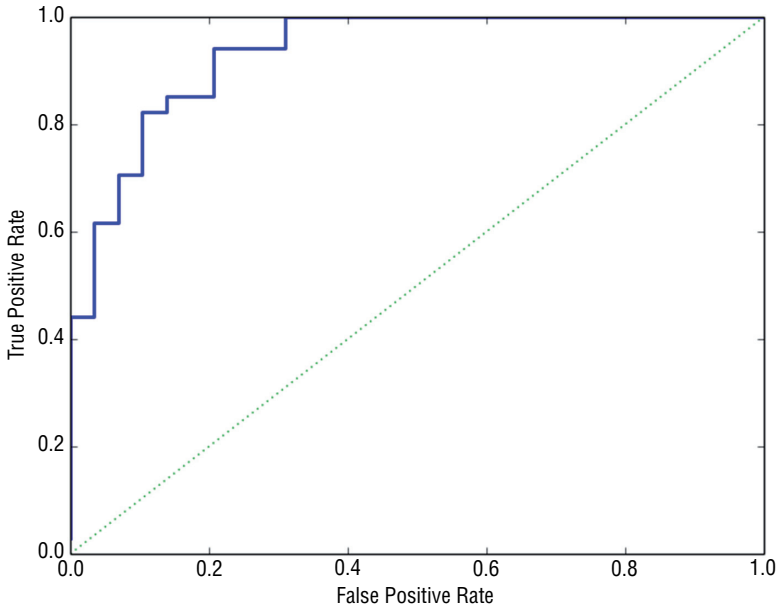

Figure 7-18: Mine detection ROC curve for Gradient Boosting

You also understand that many of the differences between building ensemble models for classification and regression stem from differences in measuring errors and otherwise characterizing errors between the two classes of problems.

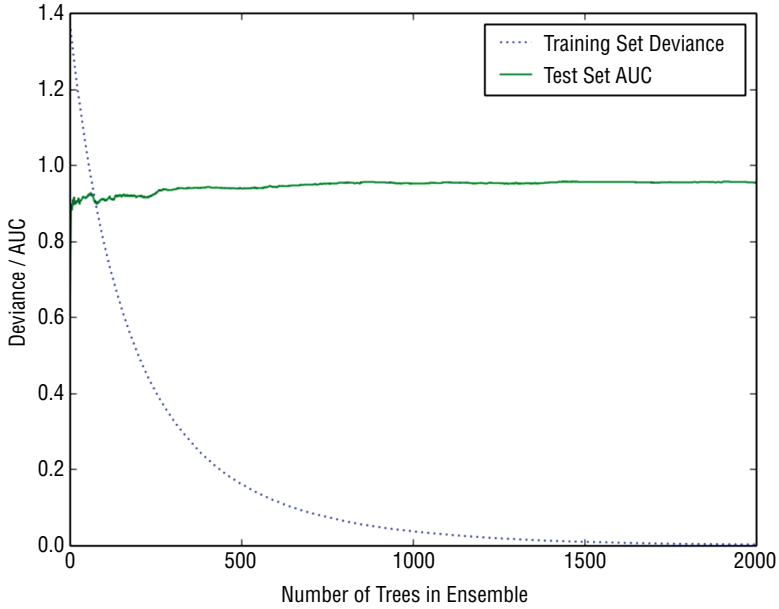

Figure 7-19: Mine detection AUC versus ensemble size for Gradient Boosting with Random Forest base learners

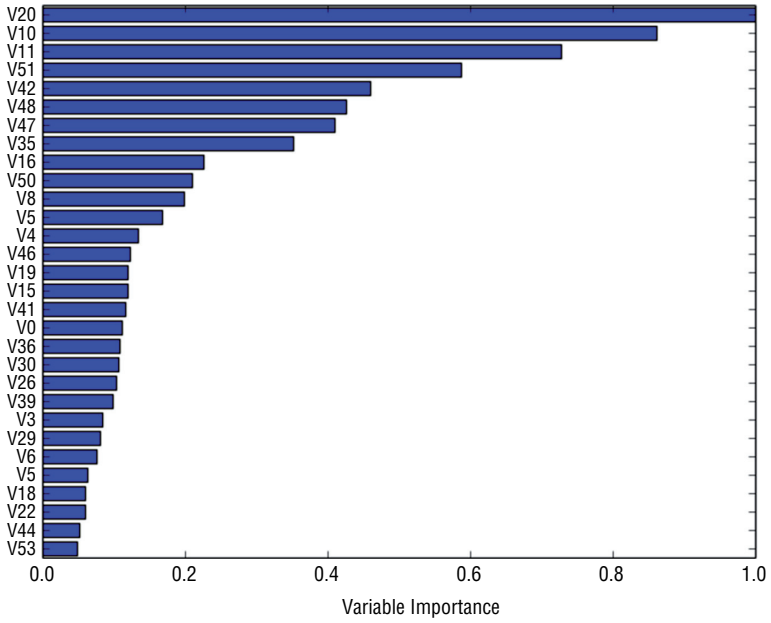

Figure 7-20: Variable importance for Gradient Boosting with Random Forest base learners

The next section shows how these methods can be used for multiclass problems.

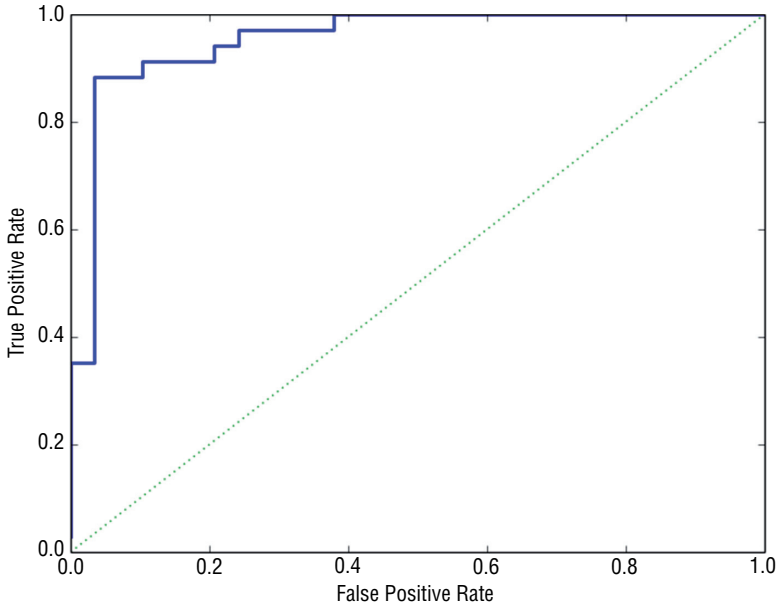

Figure 7-21: Mine detection ROC curve for Gradient Boosting using Random Forest base learners

Solving Multiclass Classification Problems with Python Ensemble Methods

The Random Forest and Gradient Boosting packages in Python will build both binary and multiclass classification models. The two types of models have a few natural differences between them. One is that the labels (`y`) take more values. The discussion of the Random Forest and Gradient Boosting packages described the manner in which the labels are specified. For a classification problem having `nClass` different classes, the labels take integer values from `0` to `nClass - 1`. Another manifestation of the number of classes is the output of the various predict methods. The predict methods that are predicting classes generate the same integer values that the labels take. The methods predicting probabilities yield probabilities for `nClass` possible classes.

The other area where there is a noticeable difference is in specifying performance. Misclassification error still makes sense, and you'll see that the example code uses that to measure oos performance. AUC is more complicated to use when there are more than two classes, and trading off different error types becomes more challenging.

Classifying Glass with Random Forests

Listing 7-9 follows a similar outline to the code used for detecting mines.

Listing 7-9: Classifying Glass Types Using Random Forests—glassRF.py

```
__author__ = 'mike_bowles'

import urllib2
from math import sqrt, fabs, exp
import matplotlib.pyplot as plot
from sklearn.linear_model import enet_path
from sklearn.metrics import accuracy_score, confusion_matrix, roc_curve
from sklearn.cross_validation import train_test_split
from sklearn import ensemble
import numpy

target_url = ("https://archive.ics.uci.edu/ml/machine-learning-"
"databases/glass/glass.data")
data = urllib2.urlopen(target_url)

#arrange data into list for labels and list of lists for attributes
xList = []
for line in data:
    #split on comma
    row = line.strip().split(",")
    xList.append(row)

glassNames = numpy.array(['RI', 'Na', 'Mg', 'Al', 'Si', 'K', 'Ca',
    'Ba', 'Fe', 'Type'])

#Separate attributes and labels
xNum = []
labels = []

for row in xList:
    labels.append(row.pop())
    l = len(row)
    #eliminate ID
    attrRow = [float(row[i]) for i in range(1, l)]
    xNum.append(attrRow)

#number of rows and columns in x matrix
nrows = len(xNum)
ncols = len(xNum[1])

#Labels are integers from 1 to 7 with no examples of 4.
#gb requires consecutive integers starting at 0
newLabels = []
labelSet = set(labels)
labelList = list(labelSet)
labelList.sort()
nlabels = len(labelList)
for l in labels:
    index = labelList.index(l)
```

continues

continued

```
    newLabels.append(index)

#Class populations:
#old label     new label     num of examples
#1              0               70
#2              1               76
#3              2               17
#5              3               13
#6              4               9
#7              5               29
#
#Drawing 30% test sample may not preserve population proportions

#stratified sampling by labels.
xTemp = [xNum[i] for i in range(nrows) if newLabels[i] == 0]
yTemp = [newLabels[i] for i in range(nrows) if newLabels[i] == 0]
xTrain, xTest, yTrain, yTest = train_test_split(xTemp, yTemp,
    test_size=0.30, random_state=531)
for iLabel in range(1, len(labelList)):
    #segregate x and y according to labels
    xTemp = [xNum[i] for i in range(nrows) if newLabels[i] == iLabel]
    yTemp = [newLabels[i] for i in range(nrows) if \
        newLabels[i] == iLabel]

    #form train and test sets on segregated subset of examples
    xTrainTemp, xTestTemp, yTrainTemp, yTestTemp = train_test_split(
        xTemp, yTemp, test_size=0.30, random_state=531)

    #accumulate
    xTrain = numpy.append(xTrain, xTrainTemp, axis=0)
    xTest = numpy.append(xTest, xTestTemp, axis=0)
    yTrain = numpy.append(yTrain, yTrainTemp, axis=0)
    yTest = numpy.append(yTest, yTestTemp, axis=0)

missCLassError = []
nTreeList = range(50, 2000, 50)
for iTrees in nTreeList:
    depth = None
    maxFeat  = 4 #try tweaking
    glassRFModel = ensemble.RandomForestClassifier(n_estimators=iTrees,
                      max_depth=depth, max_features=maxFeat,
                      oob_score=False, random_state=531)

    glassRFModel.fit(xTrain,yTrain)

    #Accumulate auc on test set
    prediction = glassRFModel.predict(xTest)
    correct = accuracy_score(yTest, prediction)

    missCLassError.append(1.0 - correct)
```

```
print("Missclassification Error" )
print(missCLassError[-1])

#generate confusion matrix
pList = prediction.tolist()
confusionMat = confusion_matrix(yTest, pList)
print('')
print("Confusion Matrix")
print(confusionMat)

#plot training and test errors vs number of trees in ensemble
plot.plot(nTreeList, missCLassError)
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Missclassification Error Rate')
#plot.ylim([0.0, 1.1*max(mseOob)])
plot.show()

# Plot feature importance
featureImportance = glassRFModel.feature_importances_

# normalize by max importance
featureImportance = featureImportance / featureImportance.max()

#plot variable importance
idxSorted = numpy.argsort(featureImportance)
barPos = numpy.arange(idxSorted.shape[0]) + .5
plot.barh(barPos, featureImportance[idxSorted], align='center')
plot.yticks(barPos, glassNames[idxSorted])
plot.xlabel('Variable Importance')
plot.show()


# Printed Output:
# Missclassification Error
# 0.227272727273
#
# Confusion Matrix
# [[17  1  2  0  0  1]
#  [ 2 18  1  2  0  0]
#  [ 3  0  3  0  0  0]
#  [ 0  0  0  4  0  0]
#  [ 0  1  0  0  2  0]
#  [ 0  2  0  0  0  7]]
```

Dealing with Class Imbalances

As stated, this listing follows a similar outline to the code used for detecting mines. There are a couple of key differences. In the code, you'll see a list of the different glass types using the numbering system from the original data set and

the corresponding integer used as labels to meet the specification required for Random Forest. The table also shows how many examples there are of each type of glass. Some of the types have relatively many examples (in the 70s). Some types of glass are not as well represented. One in particular only has nine examples.

Imbalanced classes can sometimes cause problems because random sampling of the underrepresented classes may result in wildly different proportions in the sample than in the original data. To avoid that, the code goes through a process called *stratified sampling*. What that means in this case is that the data are segregated according to labels (stratified), and then each of those groups is sampled to obtain training and test sets within each class. Then the class-specific training sets are combined into a training set that has proportions of different classes that exactly match the original data.

The code generates Random Forest models and plots the training progress and the variable importance. It also prints out a confusion matrix that shows for each true class how many of the class were predicted to be each class. If the classifier is perfect, there should be no off-diagonal entries in the matrix.

Figure 7-22 shows how the performance of Random Forests improves as more trees are included in the ensemble. The curve generally drops as more trees are added. The rate of improvement decreases as more trees are added. It has slowed considerably at the point where the graph stops.

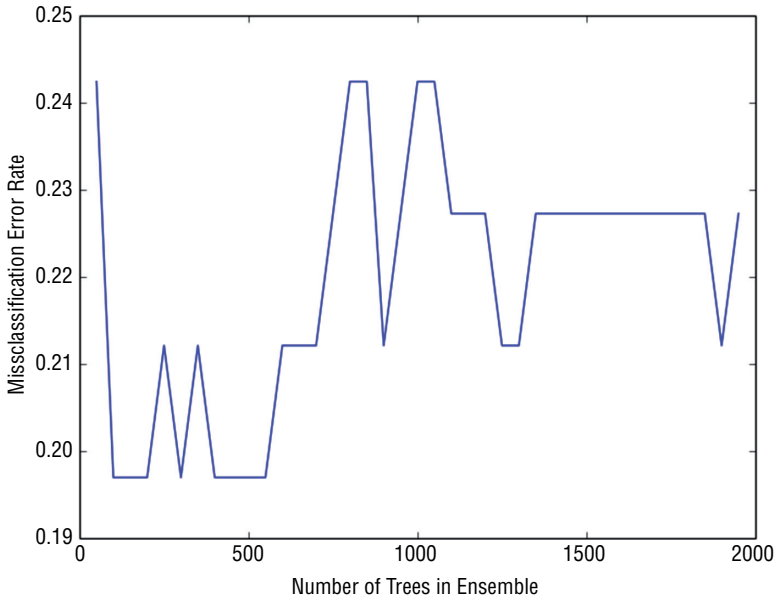

Figure 7-22: The overall performance of Random Forests

Figure 7-23 is a bar chart showing the relative importance of the variables used by Random Forests. The chart shows that a number of the variables are roughly

equal in performance. This is unusual behavior. In many cases the variable importances drop off quickly after the first few variables. In this problem there are several equally important variables.

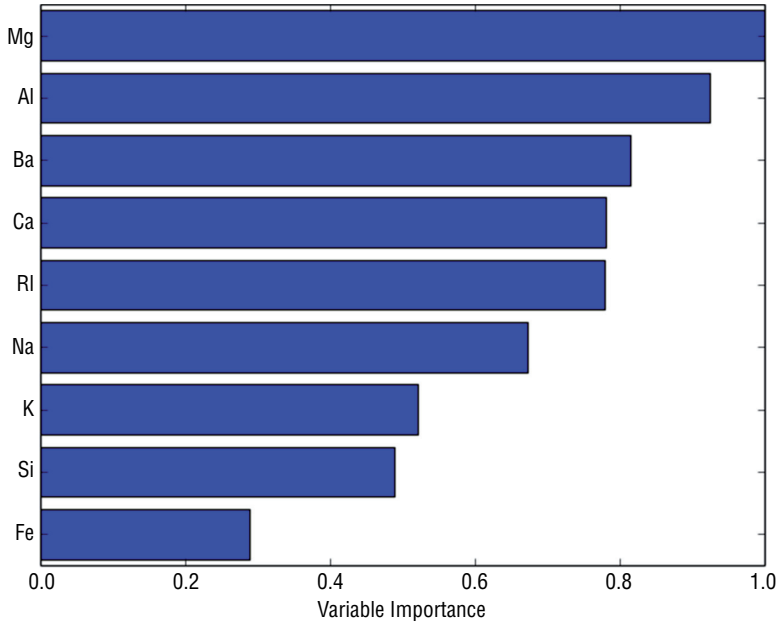

Figure 7-23: The relative importance of the variables used by Random Forest

Classifying Glass Using Gradient Boosting

Listing 7-10 runs through the same steps as the Random Forest glass classifier in the preceding section, with a couple of minor differences.

Listing 7-10: Classifying Glass with Gradient Boosting—glassGbm.py

```
__author__ = 'mike_bowles'

import urllib2
from math import sqrt, fabs, exp
import matplotlib.pyplot as plot
from sklearn.linear_model import enet_path
from sklearn.metrics import roc_auc_score, roc_curve, confusion_matrix
from sklearn.cross_validation import train_test_split
from sklearn import ensemble
import numpy


target_url = ("https://archive.ics.uci.edu/ml/machine-learning-"
```

continues

continued

```
"databases/glass/glass.data")
data = urllib2.urlopen(target_url)

#arrange data into list for labels and list of lists for attributes
xList = []
for line in data:
    #split on comma
    row = line.strip().split(",")
    xList.append(row)

glassNames = numpy.array(['RI', 'Na', 'Mg', 'Al', 'Si', 'K', 'Ca',
    'Ba', 'Fe', 'Type'])

#Separate attributes and labels
xNum = []
labels = []

for row in xList:
    labels.append(row.pop())
    l = len(row)
    #eliminate ID
    attrRow = [float(row[i]) for i in range(1, l)]
    xNum.append(attrRow)

#number of rows and columns in x matrix
nrows = len(xNum)
ncols = len(xNum[1])

#Labels are integers from 1 to 7 with no examples of 4.
#gb requires consecutive integers starting at 0
newLabels = []
labelSet = set(labels)
labelList = list(labelSet)
labelList.sort()
nlabels = len(labelList)
for l in labels:
    index = labelList.index(l)
    newLabels.append(index)

#Class populations:
#old label     new label     num of examples
#1              0              70
#2              1              76
#3              2              17
#5              3              13
#6              4              9
#7              5              29
#
#Drawing 30% test sample may not preserve population proportions

#stratified sampling by labels.
```

```
xTemp = [xNum[i] for i in range(nrows) if newLabels[i] == 0]
yTemp = [newLabels[i] for i in range(nrows) if newLabels[i] == 0]
xTrain, xTest, yTrain, yTest = train_test_split(xTemp, yTemp,
    test_size=0.30, random_state=531)
for iLabel in range(1, len(labelList)):
    #segregate x and y according to labels
    xTemp = [xNum[i] for i in range(nrows) if newLabels[i] == iLabel]
    yTemp = [newLabels[i] for i in range(nrows) if \
        newLabels[i] == iLabel]

    #form train and test sets on segregated subset of examples
    xTrainTemp, xTestTemp, yTrainTemp, yTestTemp = train_test_split(
        xTemp, yTemp, test_size=0.30, random_state=531)

    #accumulate
    xTrain = numpy.append(xTrain, xTrainTemp, axis=0)
    xTest = numpy.append(xTest, xTestTemp, axis=0)
    yTrain = numpy.append(yTrain, yTrainTemp, axis=0)
    yTest = numpy.append(yTest, yTestTemp, axis=0)

#instantiate model
nEst = 500
depth = 3
learnRate = 0.003
maxFeatures = 3
subSamp = 0.5
glassGBMModel = ensemble.GradientBoostingClassifier(n_estimators=nEst,
                    max_depth=depth,learning_rate=learnRate,
                    max_features=maxFeatures,subsample=subSamp)

#train
glassGBMModel.fit(xTrain, yTrain)

# compute auc on test set as function of ensemble size
missClassError = []
missClassBest = 1.0
predictions = glassGBMModel.staged_decision_function(xTest)
for p in predictions:
    missClass = 0
    for i in range(len(p)):
        listP = p[i].tolist()
        if listP.index(max(listP)) != yTest[i]:
            missClass += 1
    missClass = float(missClass)/len(p)

    missClassError.append(missClass)

    #capture best predictions
    if missClass < missClassBest:
        missClassBest = missClass
        pBest = p
```

continues

continued

```
idxBest = missClassError.index(min(missClassError))

#print best values
print("Best Missclassification Error" )
print(missClassBest)
print("Number of Trees for Best Missclassification Error")
print(idxBest)

#plot training deviance and test auc's vs number of trees in ensemble
missClassError = [100*mce for mce in missClassError]
plot.figure()
plot.plot(range(1, nEst + 1), glassGBMModel.train_score_,
    label='Training Set Deviance', linestyle=":")
plot.plot(range(1, nEst + 1), missClassError, label='Test Set Error')
plot.legend(loc='upper right')
plot.xlabel('Number of Trees in Ensemble')
plot.ylabel('Deviance / Classification Error')
plot.show()

# Plot feature importance
featureImportance = glassGBMModel.feature_importances_

# normalize by max importance
featureImportance = featureImportance / featureImportance.max()

#plot variable importance
idxSorted = numpy.argsort(featureImportance)
barPos = numpy.arange(idxSorted.shape[0]) + .5
plot.barh(barPos, featureImportance[idxSorted], align='center')
plot.yticks(barPos, glassNames[idxSorted])
plot.xlabel('Variable Importance')
plot.show()

#generate confusion matrix for best prediction.
pBestList = pBest.tolist()
bestPrediction = [r.index(max(r)) for r in pBestList]
confusionMat = confusion_matrix(yTest, bestPrediction)
print('')
print("Confusion Matrix")
print(confusionMat)


# Printed Output:
#
# nEst = 500
# depth = 3
# learnRate = 0.003
# maxFeatures = None
# subSamp = 0.5
#
```

```
#
# Best Missclassification Error
# 0.242424242424
# Number of Trees for Best Missclassification Error
# 113
#
# Confusion Matrix
# [[19  1  0  0  0  1]
#  [ 3 19  0  1  0  0]
#  [ 4  1  0  0  1  0]
#  [ 0  3  0  1  0  0]
#  [ 0  0  0  0  3  0]
#  [ 0  1  0  1  0  7]]
#



# For Gradient Boosting using Random Forest base learners
# nEst = 500
# depth = 3
# learnRate = 0.003
# maxFeatures = 3
# subSamp = 0.5
#
#
#
# Best Missclassification Error
# 0.227272727273
# Number of Trees for Best Missclassification Error
# 267
#
# Confusion Matrix
# [[20  1  0  0  0  0]
#  [ 3 20  0  0  0  0]
#  [ 3  3  0  0  0  0]
#  [ 0  4  0  0  0  0]
#  [ 0  0  0  0  3  0]
#  [ 0  2  0  0  0  7]]
```

As before, the Gradient Boosting version uses the "staged" methods available in the `GradientBoostingClassifier` class to generate predictions at each step in the Gradient Boosting training process.

Assessing the Advantage of Using Random Forest Base Learners with Gradient Boosting

At the end of the code, you'll see results reported for both Gradient Boosting with `max_features=None` and for `max_features=20`. The first parameter setting trains ordinary trees as suggested in the original Gradient Boosting papers.

The second parameter setting incorporates trees like the ones used in Random Forest, where not all the features are considered for splitting at each node. Instead of all the features being considered, `max_features` are selected at random for consideration as the splitting variable. This gives a sort of hybrid between the usual Gradient Boosting implementation and Random Forest.

Figure 7-24 plots the deviance on the training set and the misclassification error on the test set. The deviance indicates the progress of the training process. Misclassification on the test set is used to determine whether the model is overfitting. The algorithm does not overfit, but it also does not improve past 200 or so trees and could be terminated sooner.

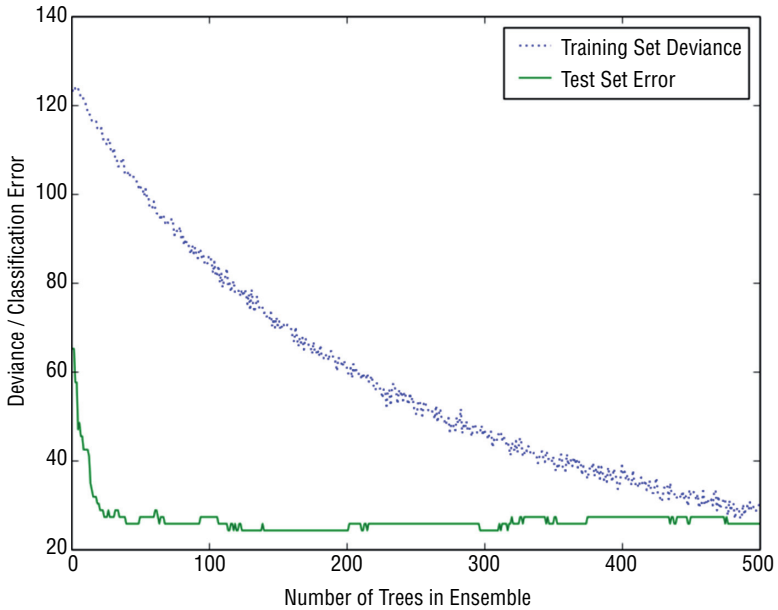

Figure 7-24: Glass classifier built using Gradient Boosting: training performance

Figure 7-25 plots the variable importance for Gradient Boosting. The variables show unusually equal importance. It's more usual to have a few variables be very important and for the importances to drop off more rapidly.

Figure 7-26 plots the deviance and oos misclassification error `max_features=20`, which results in Random Forest base learners being used in the ensemble, as discussed earlier. This leads to an improvement of about 10% in the misclassification error rate. That's not really perceptible from the graph in Figure 7-26, and the slight improvement in the end number does not change the basic character of the plot.

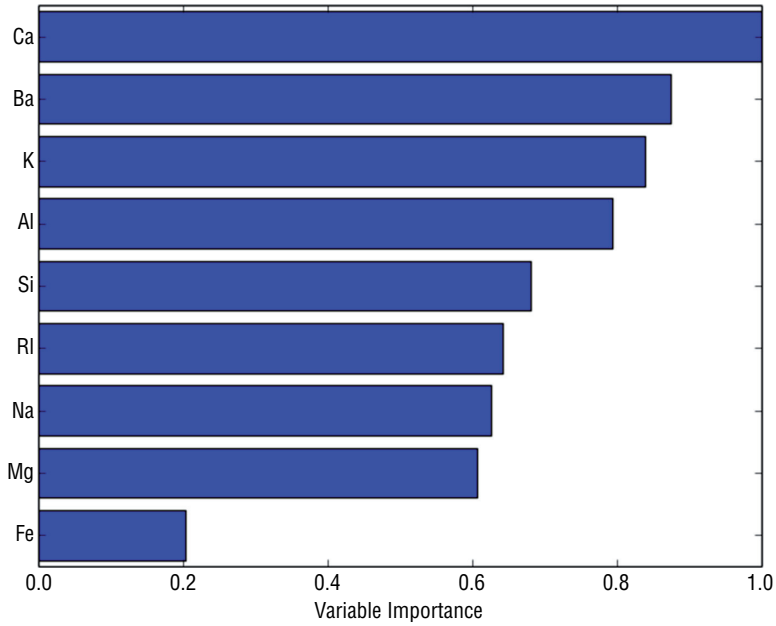

Figure 7-25: Glass classifier built using Gradient Boosting: variable importance

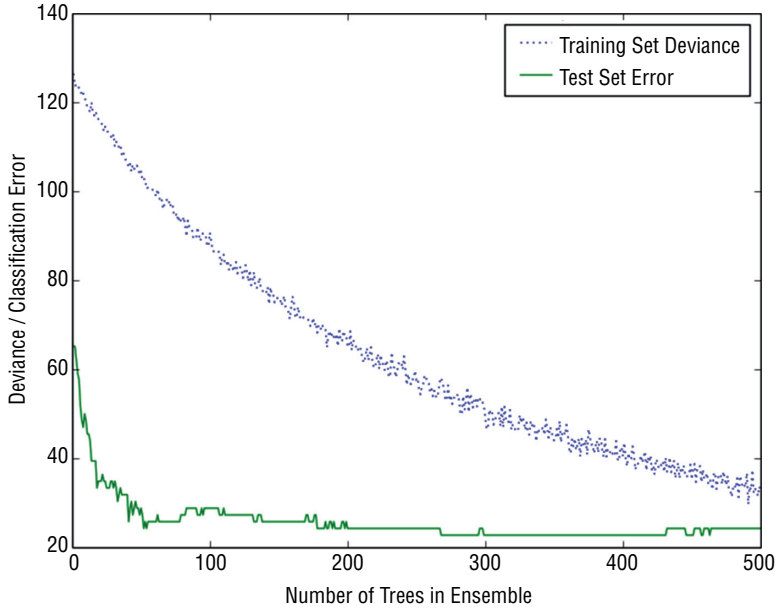

Figure 7-26: Glass classifier built using Gradient Boosting with Random Forest base learners: training performance.

Figure 7-27 shows the plot of variable importance for Gradient Boosting with Random Forest base learners. The order between this figure and Figure 7-25 is

somewhat altered. Some of the same variables appear in the top five, but some other in the top five for one are in the bottom for the other. These plots both show a surprisingly uniform level of importance, and that may be the cause of the instability in the importance order between the two.

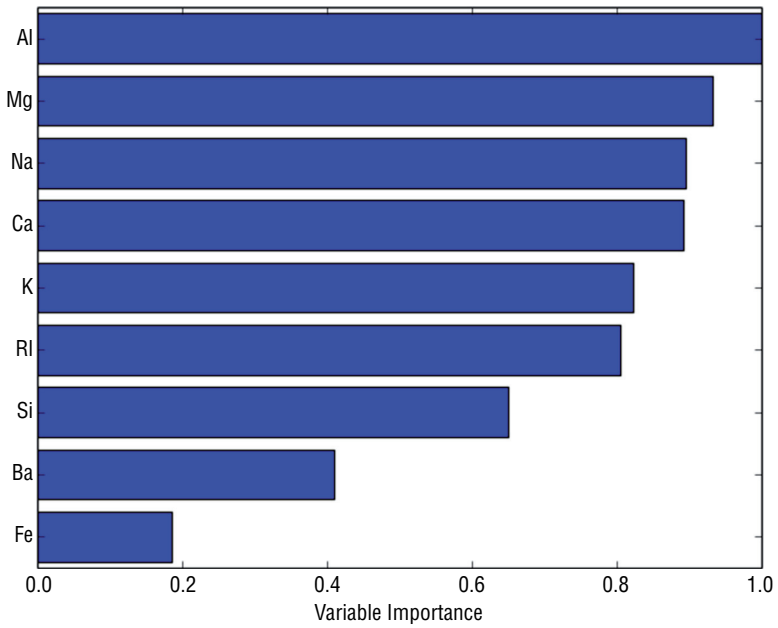

Figure 7-27: Glass classifier built using Gradient Boosting with Random Forest base learners: variable importance

Comparing Algorithms

Table 7-1 gives timing and performance comparisons for the algorithms presented here. The times shown are the training times for one complete pass through training. Some of the code for training Random Forest trained a series of different-sized models. In that case, only the last (and longest) training pass is counted. The others were done to illustrate the behavior as a function of the number of trees in the training set. Similarly, for penalized linear regression, many of the runs incorporated 10-fold cross-validation, whereas other examples used a single holdout set. The single holdout set requires one training pass, whereas 10-fold cross-validation requires 10 training passes. For examples that incorporated 10-fold cross-validation, the time for 1 of the 10 training passes is shown.

Except for the glass data set (a multiclass classification problem), the training times for penalized linear regression are an order of magnitude faster

than Gradient Boosting and Random Forest. Generally, the performance with Random Forest and Gradient Boosting is superior to penalized linear regression. Penalized linear regression is somewhat close on some of the data sets. Getting close on the wine data required employing basis expansion. Basis expansion was not used on other data sets and might lead to some further improvement.

Table 7-1: Performance and Training Time Comparisons

DATA SET	ALGORITHM	TRAIN TIME	PERFORMANCE	PERF METRIC
glass	RF 2000 trees	2.354401	0.227272727273	class error
glass	gbm 500 trees	3.879308	0.227272727273	class error
glass	lasso	12.296948	0.373831775701	class error
rvmines	rf 2000 trees	2.760755	0.950304259635	auc
rvmines	gbm 2000 trees	4.201122	0.956389452333	auc
rvmines	enet	0.519870*	0.868672796508	auc
abalone	rf 500 trees	8.060850	4.30971555911	mse
abalone	gbm 2000 trees	22.726849	4.22969363284	mse
wine	rf 500 trees	2.665874	0.314125711509	mse
wine	gbm 2000 trees	13.081342	0.313361215728	mse
wine	lasso-expanded	0.646788*	0.434528740430	mse

*The times marked with an asterisk are time per cross-validation fold. These techniques were trained several times in repetition in accordance with the n-fold cross-validation technique whereas other methods were trained using a single holdout test set. Using the time per cross-validation fold puts the comparisons on the same footing.

Random Forest and Gradient Boosting have very close performance to one another, although sometimes one or the other of them requires more trees than the other to achieve it. The training times for Random Forest and Gradient Boosting are roughly equivalent. In some of the cases where they differ, one of them is getting trained much longer than required. In the abalone data set, for example, the oos error has flattened by 1,000 steps (trees), but training continues until 2,000. Changing that would cut the training time for Gradient Boosting in half and bring the training times for that data set more into agreement. The same is true for the wine data set.

Summary

This chapter demonstrated ensemble methods available as Python packages. The examples show these methods at work building models on a variety of different types of problems. The chapter also covered regression, binary classification,

and multiclass classification problems, and discussed variations on these themes such as the workings of coding categorical variables for input to Python ensemble methods and stratified sampling. These examples cover many of the problem types that you're likely to encounter in practice.

The examples also demonstrate some of the important features of ensemble algorithms—the reasons why they are a first choice among data scientists. Ensemble methods are relatively easy to use. They do not have many parameters to tune. They give variable importance data to help in the early stages of model development, and they very often give the best performance achievable.

The chapter demonstrated the use of available Python packages. The background given in Chapter 6 helps you to understand the parameters and adjustments that you see in the Python packages. Seeing them exercised in the example code can help you get started using these packages.

The comparisons given at the end of the chapter demonstrate how these algorithms compare. The ensemble methods frequently give the best performance. The penalized regression methods are blindingly much faster than ensemble methods and in some cases yield similar performance.

References

1. sklearn documentation for RandomForestRegressor, `http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html`
2. Leo Breiman. (2001). "Random Forests." *Machine Learning*, *45*(1): 5–32. doi:10.1023/A:1010933404324
3. J. H. Friedman. "Greedy Function Approximation: A Gradient Boosting Machine," `https://statweb.stanford.edu/~jhf/ftp/trebst.pdf`
4. sklearn documentation for RandomForestRegressor, `http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html`
5. L. Breiman, "Bagging predictors," `http://statistics.berkeley.edu/sites/default/files/tech-reports/421.pdf`
6. Tin Ho. (1998). "The Random Subspace Method for Constructing Decision Forests." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *20*(8): 832–844. doi:10.1109/34.709601
7. J. H. Friedman. "Greedy Function Approximation: A Gradient Boosting Machine," `https://statweb.stanford.edu/~jhf/ftp/trebst.pdf`

8. J. H. Friedman. "Stochastic Gradient Boosting," https://statweb.stanford.edu/~jhf/ftp/stobst.pdf
9. sklearn documentation for GradientBoostingRegressor, http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html
10. J. H. Friedman. "Greedy Function Approximation: A Gradient Boosting Machine," https://statweb.stanford.edu/~jhf/ftp/trebst.pdf
11. J. H. Friedman. "Stochastic Gradient Boosting," https://statweb.stanford.edu/~jhf/ftp/stobst.pdf
12. J. H. Friedman. "Stochastic Gradient Boosting," https://statweb.stanford.edu/~jhf/ftp/stobst.pdf
13. sklearn documentation for RandomForestClassifier, http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
14. sklearn documentation for GradientBoostingClassifier, http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html