CHAPTER
5

Building Predictive Models Using Penalized Linear Methods

Chapter 2 looked at a number of different data sets with an eye toward understanding the data sets, the relations between the various attributes and labels, and the nature of the problems being posed. This chapter picks those data sets up once again and runs through some case studies demonstrating the process of building predictive models by using the penalized linear methods that you saw in Chapter 4, "Penalized Linear Regression." Generally, the model-building will be segmented into two or more parts.

You'll recall from Chapter 4 that model building with penalized linear regression has two steps. One is to train on the whole data set to trace out coefficient curves. The other is to run cross-validation to determine the best achievable out-of-sample performance and to identify the model that achieves it. The step of determining the achievable performance encompasses the hard design work, and in many of the examples in this chapter, that's the only step that will be presented. The purpose of training on the whole data set is to get the best estimates of the model coefficients. But it does not change your estimate of the errors, which are the gauge of performance.

This chapter runs through a variety of different types of problems: regression problems, classification problems, problems with categorical attributes, and problems with nonlinear dependence of the labels on the attributes. It looks at basis expansion to see whether it improves the prediction performance. In each case, the objective is to work through the steps you'd take to arrive at a deployable linear model and to consider some alternative paths so that you can ensure that you're getting all the performance you can.

Python Packages for Penalized Linear Regression

The examples in Chapter 4 used Python versions of the training algorithms involved: LARS, and coordinate descent with the ElasticNet penalty. The purpose for using the Python code in Chapter 4 was to expose the workings of the algorithms to further your understanding of them. Fortunately, you don't have to code those algorithms each time you want to use them.

Scikit-learn has packages implementing Lasso, LARS, and ElasticNet regression. There are several advantages to using those packages. One advantage is that using them results in fewer lines of code that you have to write and debug. Another big advantage is that they are much faster than the code in Chapter 4. The scikit-learn packages take advantage of practices like not computing correlations for attributes that aren't being used in order to cut way down on the number of calculations. You'll see when you run these packages that they execute very quickly.

The packages used in this chapter are found in sklearn.linear_model. The link `http://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model` shows a list including the models you'll see used here. Notice that several of the models come in two flavors. For example, there's a package titled linear_model.ElasticNet and one titled linear_model.ElasticNetCV. These two models correspond to the two tasks discussed at the beginning of this chapter. The Python package linear_model.ElasticNet is used to calculate coefficient curves on the whole data set, and linear_model.ElasticNetCV does the cross-validation run to produce out-of-sample estimates of performance. It's handy to have these two forms.

The same basic input objects fuel both versions (two numpy arrays—one of attributes and one of labels). In some cases, you won't be able to use the cross-validation version because you'll need very specific control of the contents of training and test sets for each fold:

- If your problem has a categorical attribute that takes one of its values very infrequently, you may need to control sampling so that the attribute is represented evenly across the folds.
- You may also need to have access to the separate fold data to compile error statistics for your problem, if you want a different error measure from the *mean squared error* (MSE) that the CV packages deliver. You might prefer *mean absolute error* (MAE) because it better matches the penalty that you'll pay for errors in your real problem.
- Another example of needing fold-by-fold access for error statistics is when you use linear regression to solve classification problems. As discussed in Chapter 3, "Predictive Model Building: Balancing Performance, Complexity, and Big Data," standard error measures for classification problems are

things such as misclassification error or *area under the ROC curve* (AUC). You'll see that case specifically in the "rocks versus mines" and "glass classification" case studies in this chapter.

There are a couple of things for you to keep in mind as you look at these packages and begin thinking about using them. One is that some of them (but not all of them) automatically normalize the attributes before fitting a model. The second thing to be aware of is that the scikit-learn packages name variables differently from Chapter 4 and Friedman's papers. Chapter 4 used the variable λ to represent the multiplier on the coefficient penalty and used the variable α to represent the proportion of Lasso penalty versus ridge penalty in the ElasticNet penalty. The scikit-learn packages use α instead of λ and l1_ratio instead of α. The text that follows switches to the notation used in the scikit-learn packages.

SOME SCIKIT-LEARN CHANGES

The scikit-learn documentation states an intention to bring all the penalized regression packages into conformance with one another by including normalization in all of them. That is in process at the time of writing this book.

Multivariable Regression: Predicting Wine Taste

As discussed in Chapter 2, "Understand the Problem by Understanding the Data," the wine taste data set comes from the UC Irvine data repository (`http://archive.ics.uci.edu/ml/datasets/Wine+Quality`).[1] The data set contains chemical analyses for 1,599 wines along with average taste scores given to each wine by a panel of wine tasters. The predictive problem is to predict the taste given the data on chemical composition. The chemical composition data consist of numeric measurements of 11 different chemical properties—alcohol content, pH and citric acid, and so on. Have a look at the exploration of these data in Chapter 2 or look at the UC Irvine page for the data set for more information.

Predicting the wine taste is a regression problem because the objective of the problem is to predict the quality score, which is an integer between 0 and 10. The data set only includes examples between 3 and 8. Because only integer scores are given, it is also possible to treat this problem as a multiclass classification problem. The multiclass problem would have six possible classifications (the integers from 3 to 8). It would ignore the order relation that exists among the various scores. (For example, 5 is a worse score than 6 and a better score than 4.) Regression is a more natural way to pose the problem because it preserves the order relationship.

Another way to think about how to pose the problem is to consider the different error measures that come with a regression problem versus a multiclass classification problem. The regression error function is the average squared error. When the true taste is 3, predicting a 5 contributes more to the cumulative error than predicting a 4. The error measure for the multiclass problem is the number of examples that get misclassified. With this error measure, if the true taste is 3, predicting a 5 or 4 contributes the same amount to the cumulative error. Regression seems more natural, but I don't know of a way to prove that it will give superior performance. The only way to know whether this is the best approach is to try both. In the section titled "Multiclass Classification: Classifying Crime Scene Glass Samples," you'll see how to handle multiclass classification problems. You can then come back and try the multiclass approach and see whether it does better or worse. What error measure will you use?

Building and Testing a Model to Predict Wine Taste

The first step in the process of building a model is to generate some out-of-sample performance numbers to see whether they're going to meet your performance requirements. Listing 5-1 shows the code to perform 10-fold cross-validation and plot the results. The first section of the code reads the data from the UCI website into a list of lists and then runs through normalization of the list of lists of attributes and the list of labels. Then the lists get converted to numpy arrays X (matrix of attributes) and Y (vector of labels). There are two versions of these definitions. In one version, the normalized lists are used. In the other, the un-normalized versions are used. You can comment out the second of the two definitions in either case and rerun the code to see what effect normalization the attributes or the labels has on the answers. A single line of code defines the number of cross-validation folds (10) and trains the model. Then the program plots the error versus α curves for each of the 10 folds and also plots the average of the 10. The three plots are shown in Figures 5-1, 5-2, and 5-3. In order, the three cases are as follows:

1. Normalized X and un-normalized Y
2. Normalized X and Y
3. Un-normalized X and Y

Listing 5-1: Using Cross-Validation to Estimate Out-of-Sample Error with Lasso Modeling Wine Taste—wineLassoCV.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import datasets, linear_model
from sklearn.linear_model import LassoCV
from math import sqrt
import matplotlib.pyplot as plot
```

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

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

#Normalize columns in x and labels
#Note: be careful about normalization.  Some penalized
#regression packages include it and some don't.
nrows = len(xList)
ncols = len(xList[0])

#calculate means and variances
xMeans = []
xSD = []
for i in range(ncols):
    col = [xList[j][i] for j in range(nrows)]
    mean = sum(col)/nrows
    xMeans.append(mean)
    colDiff = [(xList[j][i] - mean) for j in range(nrows)]
    sumSq = sum([colDiff[i] * colDiff[i] for i in range(nrows)])
    stdDev = sqrt(sumSq/nrows)
    xSD.append(stdDev)

#use calculate mean and standard deviation to normalize xList
xNormalized = []
for i in range(nrows):
    rowNormalized = [(xList[i][j] - xMeans[j])/xSD[j] \
        for j in range(ncols)]
    xNormalized.append(rowNormalized)

#Normalize labels
meanLabel = sum(labels)/nrows
```

continues

continued

```
sdLabel = sqrt(sum([(labels[i] - meanLabel) * (labels[i] -
        meanLabel) for i in range(nrows)])/nrows)

labelNormalized = [(labels[i] - meanLabel)/sdLabel \
        for i in range(nrows)]

#Convert list of list to np array for input to sklearn packages

#Unnormalized labels
Y = numpy.array(labels)

#normalized lables
Y = numpy.array(labelNormalized)

#Unnormalized X's
X = numpy.array(xList)

#Normalized Xss
X = numpy.array(xNormalized)

#Call LassoCV from sklearn.linear_model
wineModel = LassoCV(cv=10).fit(X, Y)

# Display results


plot.figure()
plot.plot(wineModel.alphas_, wineModel.mse_path_, ':')
plot.plot(wineModel.alphas_, wineModel.mse_path_.mean(axis=-1),
         label='Average MSE Across Folds', linewidth=2)
plot.axvline(wineModel.alpha_, linestyle='--',
            label='CV Estimate of Best alpha')
plot.semilogx()
plot.legend()
ax = plot.gca()
ax.invert_xaxis()
plot.xlabel('alpha')
plot.ylabel('Mean Square Error')
plot.axis('tight')
plot.show()

#print out the value of alpha that minimizes the Cv-error
print("alpha Value that Minimizes CV Error  ",wineModel.alpha_)
print("Minimum MSE  ", min(wineModel.mse_path_.mean(axis=-1)))

Printed Output: Normalized X, Un-normalized Y
('alpha Value that Minimizes CV Error  ', 0.010948337166040082)
('Minimum MSE  ', 0.433801987153697)

Printed Output: Normalized X and Y
('alpha Value that Minimizes CV Error  ', 0.013561387700964642)
('Minimum MSE  ', 0.66558492060028562)
```

```
Printed Output: Un-normalized X and Y
('alpha Value that Minimizes CV Error  ', 0.0052692947038249062)
('Minimum MSE  ', 0.43936035436777832)
```

The printed output at the bottom of Listing 5-1 shows a significant increase in the MSE that comes with normalizing Y. In contrast, Figures 5-1 and 5-2 are remarkably similar in shape. The only difference between them is the scale on the Y-axis. Refer to Listing 2-13 to see that the standard deviation of the unscaled wine quality scores is roughly 0.81. That means that the normalization to a standard deviation of 1.0 requires multiplying by roughly 1.2. That results in an increase of 1.2 squared in the MSE. The only issue with normalizing the labels is that the MSE loses its connection to the original data. It's usually handier to be able to extract a square root of the MSE and then relate it directly to the units of the original labels. In this case, the MSE (with un-normalized Y) is 0.433. The square root is roughly 0.65. That means that the +/- 1-sigma errors lie in a band that's 1.3 units of taste-score wide. So, normalizing Y doesn't make a material difference in the results. What about normalizing X? Does normalizing X improve or worsen performance?

The last set of numbers in Listing 5-1 shows a very slight increase in the MSE if X is left un-normalized. However, the plot of CV error versus alpha in Figure 5-3 shows a radical difference from the plots in Figures 5-1 and 5-2. The plot has a scalloped character that's caused by the mishmash of scales that comes from leaving the Xs unscaled. What happens is that the algorithm picks a large variable that requires a correspondingly small coefficient. That can happen if the variable has high correlation with Y or if the variable has low correlation with Y and a large scale. The algorithm uses a somewhat inferior variable for a few iterations until α (formerly known as λ) gets small enough to let in a better variable, at which time the error drops precipitously. The moral of the story is to normalize the Xs or be wary about not normalizing them.

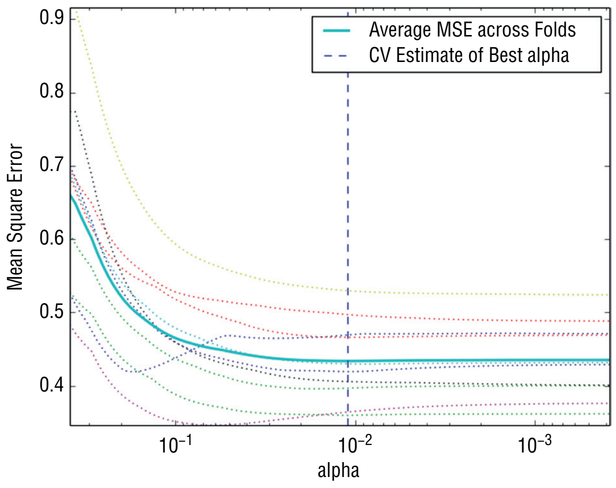

Figure 5-1: Out-of-sample error with un-normalized Y – Lasso model on wine taste data

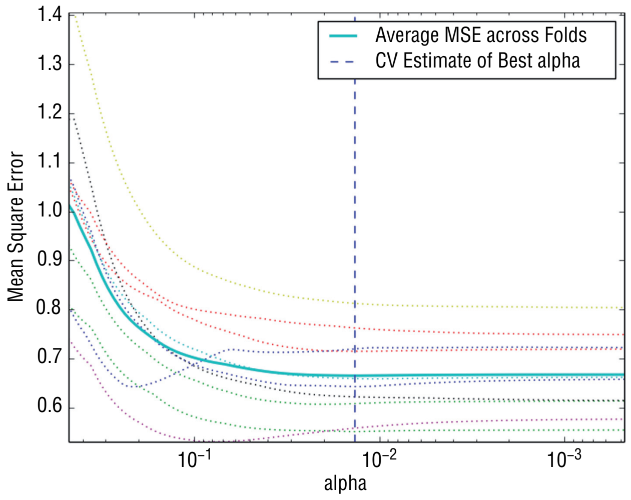

Figure 5-2: Out-of-sample error with normalized Y – Lasso model on wine taste data

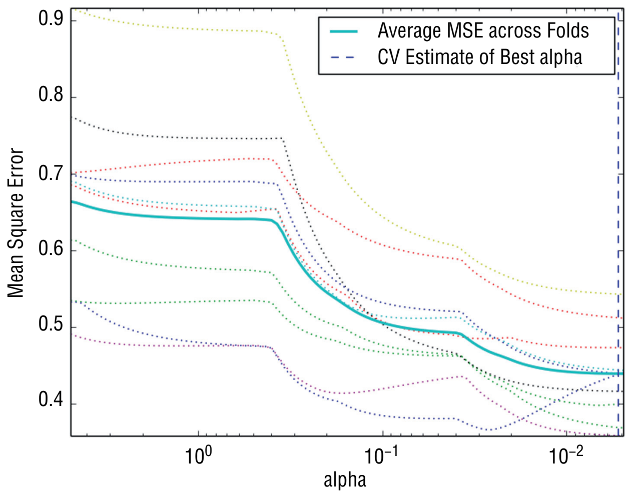

Figure 5-3: Out-of-sample error with un-normalized X and Y – Lasso model on wine taste data

Training on the Whole Data Set before Deployment

Listing 5-2 shows the code for training on the whole data set. As mentioned, the reason for training on the whole data set is to obtain the best set of coefficients for deployment. Cross-validation yields an estimate of the deployed model's performance and gives you the α value that yields the best performance. After reading the wine data from the UC Irvine data repository and normalizing it, the program converts the data to numpy arrays and then invokes the `lasso_path` method to generate α values (that is, penalties) and the corresponding coefficients. Those coefficient trajectories are plotted in Figure 5-4.

Listing 5-2: Lasso Training on Full Data Set—wineLassoCoefCurves.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import datasets, linear_model
from sklearn.linear_model import LassoCV
from math import sqrt
import matplotlib.pyplot as plot

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

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

#Normalize columns in x and labels
#Note: be careful about normalization.  Some penalized regression
#packages include it and some don't.

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

#calculate means and variances
xMeans = []
xSD = []
for i in range(ncols):
    col = [xList[j][i] for j in range(nrows)]
```

continues

continued

```
    mean = sum(col)/nrows
    xMeans.append(mean)
    colDiff = [(xList[j][i] - mean) for j in range(nrows)]
    sumSq = sum([colDiff[i] * colDiff[i] for i in range(nrows)])
    stdDev = sqrt(sumSq/nrows)
    xSD.append(stdDev)

#use calculate mean and standard deviation to normalize xList
xNormalized = []
for i in range(nrows):
    rowNormalized = [(xList[i][j] - xMeans[j])/xSD[j] for j in
    range(ncols)]
    xNormalized.append(rowNormalized)

#Normalize labels
meanLabel = sum(labels)/nrows
sdLabel = sqrt(sum([(labels[i] - meanLabel) * (labels[i] - meanLabel)
for i in range(nrows)])/nrows)

labelNormalized = [(labels[i] - meanLabel)/sdLabel for i in
range(nrows)]

#Convert list of list to np array for input to sklearn packages

#Unnormalized labels
Y = numpy.array(labels)

#normalized lables
Y = numpy.array(labelNormalized)

#Unnormalized X's
X = numpy.array(xList)

#Normalized Xss
X = numpy.array(xNormalized)

alphas, coefs, _  = linear_model.lasso_path(X, Y,  return_models=False)


plot.plot(alphas,coefs.T)

plot.xlabel('alpha')
plot.ylabel('Coefficients')
plot.axis('tight')
plot.semilogx()
ax = plot.gca()
ax.invert_xaxis()
plot.show()

nattr, nalpha = coefs.shape
```

```
#find coefficient ordering
nzList = []
for iAlpha in range(1,nalpha):
    coefList = list(coefs[: ,iAlpha])
    nzCoef = [index for index in range(nattr) if coefList[index] != 0.0]
    for q in nzCoef:
        if not(q in nzList):
            nzList.append(q)

nameList = [names[nzList[i]] for i in range(len(nzList))]
print("Attributes Ordered by How Early They Enter the Model", nameList)

#find coefficients corresponding to best alpha value. alpha value
# corresponding to normalized X and normalized Y is 0.013561387700964642

alphaStar = 0.013561387700964642
indexLTalphaStar = [index for index in range(100) if alphas[index] >
alphaStar]
indexStar = max(indexLTalphaStar)

#here's the set of coefficients to deploy
coefStar = list(coefs[:,indexStar])
print("Best Coefficient Values ", coefStar)

#The coefficients on normalized attributes give another slightly
#different ordering

absCoef = [abs(a) for a in coefStar]

#sort by magnitude
coefSorted = sorted(absCoef, reverse=True)

idxCoefSize = [absCoef.index(a) for a in coefSorted if not(a == 0.0)]

namesList2 = [names[idxCoefSize[i]] for i in range(len(idxCoefSize))]

print("Attributes Ordered by Coef Size at Optimum alpha", namesList2)



Printed Output w. Normalized X:
('Attributes Ordered by How Early They Enter the Model',
['"alcohol"', '"volatile acidity"', '"sulphates"',
'"total sulfur dioxide"', '"chlorides"', '"fixed acidity"', '"pH"',
'"free sulfur dioxide"', '"residual sugar"', '"citric acid"',
 '"density"'])

('Best Coefficient Values ',
[0.0, -0.22773815784738916, -0.0, 0.0, -0.094239023363375404,
0.022151948563542922, -0.099036391332770576, -0.0,
```

continues

continued

```
-0.067873612822590218, 0.16804102141830754, 0.37509573430881538])

('Attributes Ordered by Coef Size at Optimum alpha',
['"alcohol"', '"volatile acidity"', '"sulphates"',
'"total sulfur dioxide"', '"chlorides"', '"pH"',
'"free sulfur dioxide"'])


Printed Output w. Un-normalized X:
('Attributes Ordered by How Early They Enter the Model',
['"total sulfur dioxide"', '"free sulfur dioxide"', '"alcohol"',
'"fixed acidity"', '"volatile acidity"', '"sulphates"'])

('Best Coefficient Values ', [0.044339055570034182, -1.0154179864549988,
0.0, 0.0, -0.0, 0.0064112885435006822, -0.0038622920281433199, -0.0,
-0.0, 0.41982634135945091, 0.37812720947996975])

('Attributes Ordered by Coef Size at Optimum alpha',
['"volatile acidity"', '"sulphates"', '"alcohol"', '"fixed acidity"',
'"free sulfur dioxide"', '"total sulfur dioxide"'])
```

The program has hard-coded the α value that gave the best results in cross-validation. The version in the code is the best alpha trained with normalized attributes and labels. Changing either of these to un-normalized will change the corresponding value of the best α. Changing Y to un-normalized changes it by the 1.2 factor that comes from normalizing the standard deviation to 1.0 (as discussed earlier in the context of the MSE difference between normalized and un-normalized labels). The hard-coded value of α is used to identify the vector of coefficients corresponding to the best cross-validation results.

Listing 5-2 shows printed output for three cases: normalized attributes and un-normalized labels, both normalized, and both un-normalized. The printed output for each case includes a list of the attributes in the order that they enter the model as α is decreased. (The α in Python packages corresponds to the penalty term λ in Chapter 4.) The printed output also shows the coefficients at the hard-coded value of α. The third element of the printed output is the order of the attributes as determined by the magnitude of the corresponding coefficient (at the hard-coded value of α). The magnitude of the coefficients is another way to determine the relative importance of attributes. This ranking only makes sense when the attributes are normalized. Observe that with normalized attributes, the two methods discussed for assigning importance to attributes (order in which they appear in the solutions and relative coefficient magnitudes) give essentially the same ordering on the attributes with some disagreement on less important attributes. With un-normalized attributes, this is far from true.

As mentioned earlier, the order in which variables come into the solution (as α decreases) is strongly modified by normalizing the attributes. If a variable isn't normalized, its scale factor determines its usage instead of its inherent value in predicting the labels. This is obvious from comparing the variable ordering for normalized attributes (the first case in the printout) to the ordering for un-normalized variables.

Figures 5-4 and 5-5 show the Lasso coefficient curves for the case of normalized attributes and un-normalized attributes, respectively. The coefficient curves for un-normalized attributes are less orderly than they are for normalized attributes. Several of the early coefficients hover near zero relative to the magnitudes of coefficients that come into play later along the coefficient trajectories. This is compatible with the radically different ordering between the order that coefficients enter the model and the magnitude of the coefficient at the best solution.

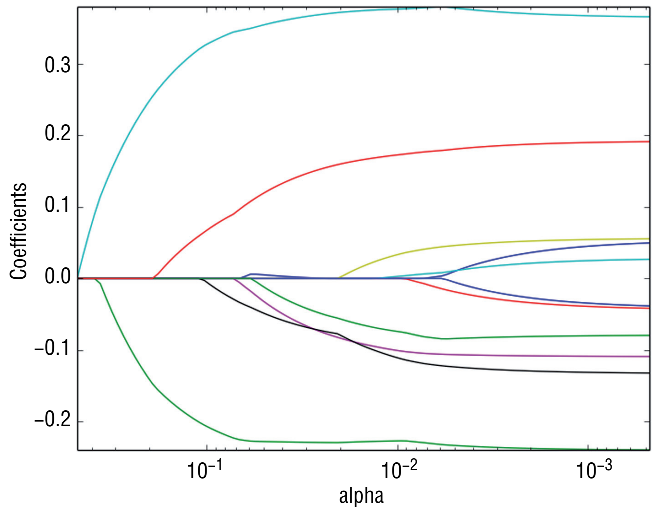

Figure 5-4: Coefficient curves for Lasso trained to predict wine quality

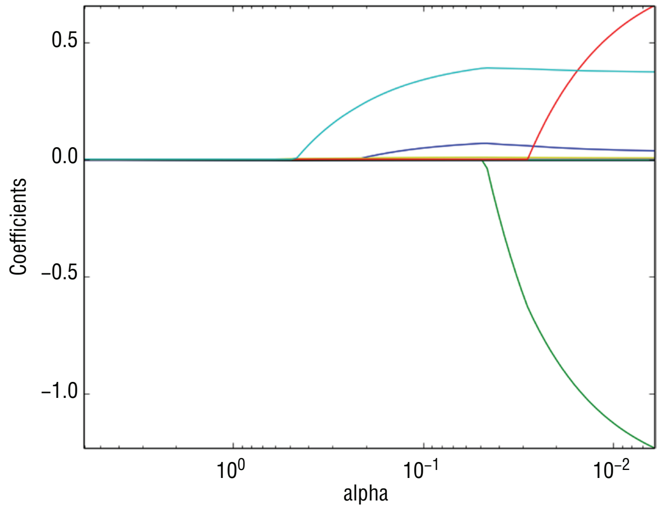

Figure 5-5: Coefficient curves for Lasso trained on un-normalized Xs

Basis Expansion: Improving Performance by Creating New Variables from Old Ones

Chapter 4 discussed adding new attributes in the form of functions of the old attributes. The point of doing that is to see whether it results in improved performance. Listing 5-3 shows how to add two new attributes to the wine data.

Listing 5-3: Using Out-of-Sample Error to Evaluate New Attributes for Predicting Wine Quality—wineExpandedLassoCV.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import datasets, linear_model
from sklearn.linear_model import LassoCV
from math import sqrt
import matplotlib.pyplot as plot

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

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

#append square of last term (alcohol)

for i in range(len(xList)):
    alcElt = xList[i][-1]
    volAcid = xList[i][1]
    temp = list(xList[i])
    temp.append(alcElt*alcElt)
    temp.append(alcElt*volAcid)
    xList[i] = list(temp)

#add new name to variable list
names[-1] = "alco^2"
names.append("alco*volAcid")

#Normalize columns in x and labels
#Note: be careful about normalization. Some penalized regression
packages include it and some don't.

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

#calculate means and variances
xMeans = []
xSD = []
for i in range(ncols):
    col = [xList[j][i] for j in range(nrows)]
    mean = sum(col)/nrows
    xMeans.append(mean)
    colDiff = [(xList[j][i] - mean) for j in range(nrows)]
    sumSq = sum([colDiff[i] * colDiff[i] for i in range(nrows)])
    stdDev = sqrt(sumSq/nrows)
    xSD.append(stdDev)

#use calculate mean and standard deviation to normalize xList
xNormalized = []
for i in range(nrows):
    rowNormalized = [(xList[i][j] - xMeans[j])/xSD[j] \
        for j in range(ncols)]
    xNormalized.append(rowNormalized)

#Normalize labels
meanLabel = sum(labels)/nrows
sdLabel = sqrt(sum([(labels[i] - meanLabel) * (labels[i] - meanLabel) \
        for i in range(nrows)])/nrows)

labelNormalized = [(labels[i] - meanLabel)/sdLabel \
        for i in range(nrows)]

#Convert list of list to np array for input to sklearn packages

#Unnormalized labels
Y = numpy.array(labels)

#normalized labels
#Y = numpy.array(labelNormalized)

#Unnormalized X's
X = numpy.array(xList)
```

continues

continued

```
#Normalized Xss
X = numpy.array(xNormalized)

#Call LassoCV from sklearn.linear_model
wineModel = LassoCV(cv=10).fit(X, Y)

# Display results


plot.figure()
plot.plot(wineModel.alphas_, wineModel.mse_path_, ':')
plot.plot(wineModel.alphas_, wineModel.mse_path_.mean(axis=-1),
         label='Average MSE Across Folds', linewidth=2)
plot.axvline(wineModel.alpha_, linestyle='--',
            label='CV Estimate of Best alpha')
plot.semilogx()
plot.legend()
ax = plot.gca()
ax.invert_xaxis()
plot.xlabel('alpha')
plot.ylabel('Mean Square Error')
plot.axis('tight')
plot.show()

#print out the value of alpha that minimizes the CV-error
print("alpha Value that Minimizes CV Error  ",wineModel.alpha_)
print("Minimum MSE  ", min(wineModel.mse_path_.mean(axis=-1)))

Printed Output: [filename - wineLassoExpandedCVPrintedOutput.txt]
('alpha Value that Minimizes CV Error  ', 0.016640498998569835)
('Minimum MSE  ', 0.43452874043020256)
```

The key step comes right after the attributes are read in and converted to floats. There are a dozen or so lines of code that take each row of attributes, pull out the two variables corresponding to measures of alcohol and volatile acidity, and then append *alcohol squared* and the product *alcohol times volatile acidity*. These are chosen because it makes sense to start with variables that are more important in the solution. A thorough hunt for possible improvements might include several attempts with combinations of the top variables.

The results show that adding these new variables degrades performance slightly. A little hunting might turn up some variables that make a useful difference. You might run out coefficient curves for this example to see whether the new variables replaced any old ones that were important at the optimum solution. That information might lead you to remove the old variables in favor of these new synthetic ones.

Figure 5-6 shows the cross-validation error curves for Lasso trained using the expanded set of attributes. The character of the cross-validation curves doesn't show substantial difference from the curves without basis expansion.

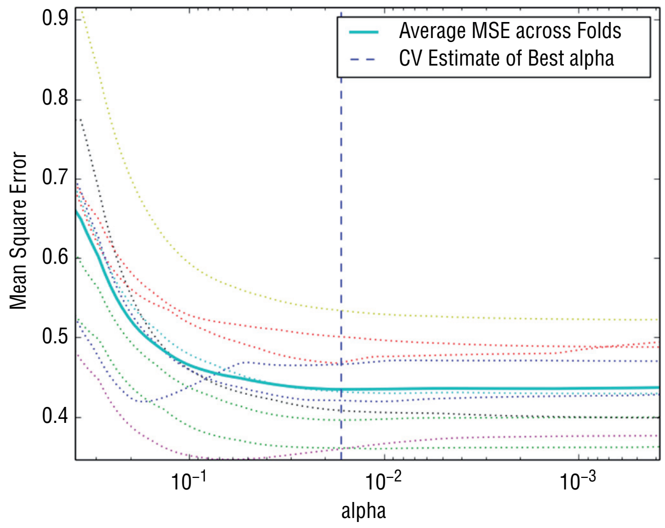

Figure 5-6: Cross-validation error curves for Lasso trained on wine quality data with expanded feature set

This section has demonstrated the use of penalized regression methods on a problem with real number outcomes—a regression problem. The next section shows the use of penalized linear regression methods on a problem where the outcomes are two-valued. The code will look similar to what you have seen in this section, and some of the techniques, like basis expansion, can be used in classification problems. The main difference is how performance is scored for a classification problem.

Binary Classification: Using Penalized Linear Regression to Detect Unexploded Mines

Chapter 4 discussed how you can use penalized linear regression for classification problems and set the process up for the rocks versus mines problem. This section gets into the details of how you would approach and solve a binary classification problem using penalized linear regression. The section incorporates the Python ElasticNet package. You'll recall from Chapter 4 that ElasticNet incorporates a more general penalty function that includes the Lasso and ridge regression penalty functions as special cases. This makes it possible to see how performance of the classifier changes as you make alterations in the penalty function. These are the steps along the path to a solution:

1. Cast the binary classification problem as a regression problem. Construct an outcome vector of real number labels by assigning 0.0 when the class outcome takes one of its two values and assigning 1.0 when it takes the other.

2. Perform cross-validation. The cross-validation becomes a little more complicated because you'll need to calculate an error quantity for each fold. Scikit-learn has some handy utilities to streamline these calculations.

The first step (outlined in Chapter 4) is to cast the binary classification problem as a regression problem by replacing the classification labels with real number labels. The rocks versus mines problem is basically to build a system using sonar to detect unexploded mines on the seabed. You'll recall from the data discovery in Chapter 2 that the data set contains digitized versions of the signals returned from rocks and from metal cylinders shaped like mines. The objective is to build a prediction system that can process the digitized signals to correctly identify whether the object is a rock or a mine. The data set consists of 208 experiments. Of the 208, 111 are mines and 97 are rocks. The data set is 61 columns wide. The first 60 columns contain the digitized sonar return. The last column contains an M or an R, depending on whether the object is in a rock or a mine. The 60 columns of numbers are the attributes for the problem. A regression problem requires numeric labels too. An approach outlined in Chapter 4 is to build the column of numeric labels by assigning the number 1 to one of the two cases and 0 to the other. Listing 5-4 initializes an empty list called *labels* and appends a 1.0 for each M row and appends a 0.0 for each R row.

With numeric attributes and numeric labels, everything is in place to use the regression version of penalized linear regression. The next logical step is to perform cross-validation to get an estimate of out-of-sample performance and identify the best value of α, the penalty parameter. For this problem, doing cross-validation requires building a cross-validation loop to enclose training and testing. Why build a cross-validation loop instead of using the cross-validation package available in Python (like the one used in the wine quality example earlier in this chapter)?

The cross-validation for regression is based on MSE. That's perfectly reasonable for a regression problem, but not for a classification problem. As discussed in Chapter 3, you characterize performance differently for a classification problem than for a regression problem. Chapter 3 discussed several ways to characterize performance. One natural way is to measure the percentage of examples that are misclassified. Another way is to measure the AUC. See Chapter 3 or the Wikipedia page `http://en.wikipedia.org/wiki/Receiver_operating_characteristic` to refresh your memory on the AUC measure. To measure either of these requires that you have access to the predictions and labels in each of the cross-validation folds. You can't judge misclassification error from a summary of the MSE for the fold.

The cross-validation loop breaks the data into training and test sets and then calls the Python `enet_path` method to accomplish training on the training portion of the data. Two inputs to the routine are different from defaults. One is the `l1_ratio`, which is set equal to `0.8`. This parameter determines what fraction of the penalty is sum of absolute values of coefficients. The value 0.8 means that penalty function is 80 percent sum of absolute values and 20 percent sum of squares. The other nondefault parameter is `fit_intercept`, which is set to `False`. The code is using normalized labels and normalized attributes. Because all of these are zero mean, there's no need to calculate an intercept term. The intercept is required only to adjust any constant offset between the attributes and the labels. Eliminating the need for the intercept term by using normalized labels makes the calculation of predictions a little cleaner. The only downside of normalizing the labels is that it makes the MSE calculation less meaningful relative to a regression problem, but for a classification problem, you're not going to use that metric of performance anyway.

In each fold, after training is completed, the coefficients that are produced are used to generate predictions on the out-of-sample data for the fold. This is accomplished in the code by using the numpy dot function, the attributes for out-of-sample data for the fold, and the coefficients for the fold. This matrix-like multiplication of two numpy arrays leads to another two-dimensional array whose rows correspond to the rows in the out-of-sample test data for the fold and whose columns correspond to the sequence of models generated by `enet_path` (that is, the sequence of coefficient vectors and the corresponding sequence of α's). These matrices of predictions for each fold are concatenated (visualize stacking them atop one another), as are the out-of-sample labels. Then, at the end of the run, these compendia of the fold-by-fold out-of-sample results can be processed easily and efficiently to yield performance data for each model and to select a model complexity (α) for deployment.

Listing 5-4 generates comparisons using two metrics. The first is misclassification error. The second is area under the *receiver operating curve* (ROC). Each column from the matrix of predictions represents predictions generated for the totality of the out-of-sample data for one set of model coefficients. All the data are represented in each column since every row is held out in one (and only one) of the folds. The misclassification comparison considers the prediction data one column at a time and out-of-sample labels (called `yOut` in the code) accumulated fold by fold. Each prediction is compared to a fixed threshold (0.0 in this example) to determine a predicted classification. Then the predicted classification is compared to the corresponding entry in `yOut` to determine whether the predicted classification is correct. The plot in Figure 5-7 shows several points that achieve the same minimum. It's good practice when you have a choice to choose the point farthest to the left on the graph of performance versus α. That's because points to the right have more tendency to be overfit. It's more conservative to choose a solution farther to the left. You'll have a better chance that the errors in deployment will match those you see in cross-validation.

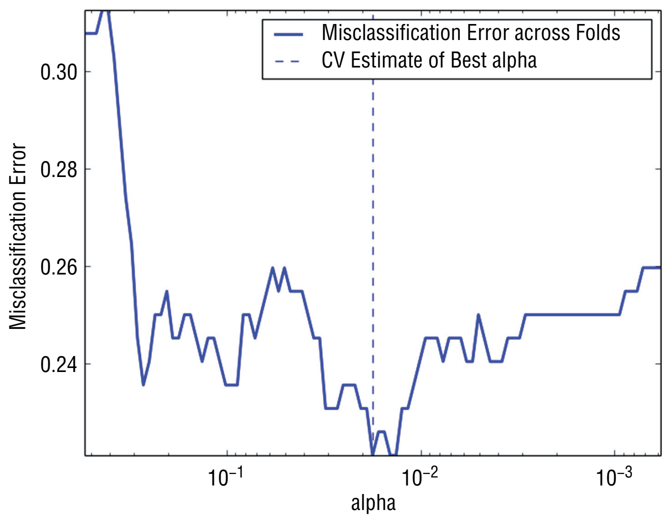

Figure 5-7: Out-of-sample classifier misclassification performance

Another way to measure the performance of your classifier is AUC. AUC has the advantage that in maximizing the AUC you wind up getting the best performance independent of where you intend to operate the system—whether you want more or less equal rates of different types of errors or you'd prefer to bias the errors toward one type. Strictly speaking, maximizing AUC does not guarantee that you'll get optimum performance at a particular error rate. Comparing the model chosen by AUC to the one chosen by minimizing overall error rate and observing the shapes of the curves help you get confidence in your solution and give you some idea about how much more performance is available with more thorough optimization.

The AUC calculations shown in Listing 5-4 use roc_curve and roc_auc_score programs from the sklearn. The process for generating the AUC versus α curve is similar to the process for the misclassification error, except the column of predictions and the true values are passed to the roc_auc_score program to generate the AUC number. Those then get plotted in Figure 5-8. The resulting curve looks roughly like the misclassification error curve upside down—upside down because larger is better for AUC, whereas smaller is better for misclassification error. The printed output at the end of Listing 5-4 shows that the location of the optimum model based on misclassification error isn't exactly the same as the optimum model for AUC, but they're not far apart. Figure 5-9 shows the ROC plot for the classifier that maximizes AUC.

In your problem, some errors might be more expensive than others, causing you to want to bias the results away from the expensive errors in favor of the less expensive errors. For the rocks versus mines problem, there may be much higher expense for incorrectly classifying an unexploded mine as a rock than for classifying a rock as a mine.

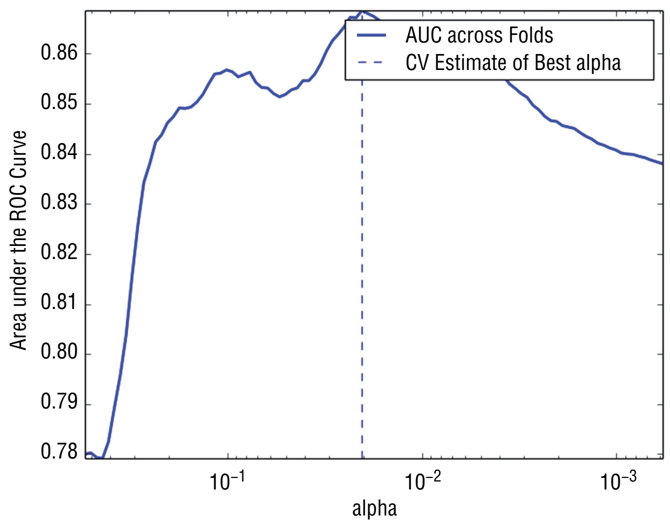

Figure 5-8: Out-of-sample classifier AUC performance

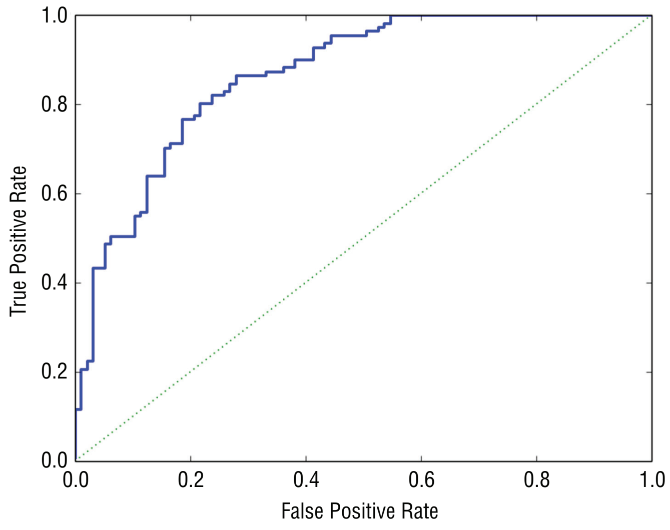

Figure 5-9: Receiver operating characteristic for best performing classifier

One systematic way to deal with this is to use a confusion matrix, discussed in Chapter 3. It's relatively easy to build from the output of the roc_curve program. The points on the ROC curve correspond to different values of threshold. The point (1,1) corresponds to the extreme where the threshold is set so low that all the points are classified as mines. That makes both the true positive rate and false positive rate equal one; the classifier gets all the positive points right, but it also gets all the negative points wrong. Setting the threshold higher than all the points gives the opposite corner of the plot. Getting the details on how points are shifting between the various boxes in the confusion matrix requires picking some threshold values and printing out the results. Listing 5-4 shows three values of threshold chosen from the range of threshold values at inner quartiles of the threshold values (that is, excluding the end points). Setting the threshold high results in low false positives and high false negatives. Setting the threshold low has the opposite behavior. Setting the threshold in the middle more nearly balances the two types of errors.

You could get a best value of threshold by associating costs with each type of error and finding the value of the threshold that minimizes the total cost. The three confusion matrices in the printed output can serve as an example for how this would work. If false positive and false negative both cost $1, the middle table (corresponding to a threshold value of -0.0455) gives a total cost of $46, whereas the higher threshold gives $68 and the lower threshold gives $54. However, if the cost for false positive is $10 and the cost for false negative is $1, the higher threshold gives $113, the middle gives $226, and the lower gives $504. You might want to test more threshold values at finer granularity. For this approach to work properly, you'll need to get the costs in a reasonable ballpark, and you'll need to make sure that the percentages of positive cases and negative cases match those that you'll see in real examples. The rocks versus mines examples were set up in a laboratory environment and probably don't represent the actual numbers of rocks versus mines in a harbor. That's easy enough to fix by oversampling one class or the other—that is, replicating some of the examples in one class or the other to get the proportions to match those you expect to see in deployment.

The data in the rocks versus mines training set are fairly well balanced. That is, there are roughly the same number of positive and negative examples. In some data sets, there may be many more examples of one class or the other. For example, clicks on Internet ads are a small fraction of 1 percent of the number of times the ads are seen. You may get better training results by over-representing the less numerous examples so that the proportions are closer to equal. You can accomplish this by replicating some of the less numerous cases or removing some of the more numerous ones.

Listing 5-4: Using ElasticNet Regression to Build a Binary (Two-Class) Classifier—rocksVMinesENetRegCV.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
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(1.0)
    else:
        labels.append(0.0)
    attrRow = [float(elt) for elt in row]
    xNum.append(attrRow)

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

alpha = 1.0

#calculate means and variances
xMeans = []
xSD = []
for i in range(ncol):
    col = [xNum[j][i] for j in range(nrow)]
    mean = sum(col)/nrow
    xMeans.append(mean)
    colDiff = [(xNum[j][i] - mean) for j in range(nrow)]
    sumSq = sum([colDiff[i] * colDiff[i] for i in range(nrow)])
    stdDev = sqrt(sumSq/nrow)
    xSD.append(stdDev)

#use calculate mean and standard deviation to normalize xNum
xNormalized = []
for i in range(nrow):
    rowNormalized = [(xNum[i][j] - xMeans[j])/xSD[j] \
        for j in range(ncol)]
    xNormalized.append(rowNormalized)

#normalize labels to center
#Normalize labels
```

continues

continued

```
meanLabel = sum(labels)/nrow
sdLabel = sqrt(sum([(labels[i] - meanLabel) * (labels[i] - meanLabel) \
        for i in range(nrow)])/nrow)

labelNormalized = [(labels[i] - meanLabel)/sdLabel for i in range(nrow)]


#number of cross-validation folds
nxval = 10


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

    #Define test and training attribute and label sets
    xTrain = numpy.array([xNormalized[r] for r in idxTrain])
    xTest = numpy.array([xNormalized[r] for r in idxTest])
    labelTrain = numpy.array([labelNormalized[r] for r in idxTrain])
    labelTest = numpy.array([labelNormalized[r] for r in idxTest])
    alphas, coefs, _ = enet_path(xTrain, labelTrain,l1_ratio=0.8,
        fit_intercept=False, return_models=False)

    #apply coefs to test data to produce predictions and accumulate
    if ixval == 0:
        pred = numpy.dot(xTest, coefs)
        yOut = labelTest
    else:
        #accumulate predictions
        yTemp = numpy.array(yOut)
        yOut = numpy.concatenate((yTemp, labelTest), axis=0)

        #accumulate predictions
        predTemp = numpy.array(pred)
        pred = numpy.concatenate((predTemp, numpy.dot(xTest, coefs)),
            axis = 0)


#calculate misclassification error
misClassRate = []
_,nPred = pred.shape
for iPred in range(1, nPred):
    predList = list(pred[:, iPred])
    errCnt = 0.0
    for irow in range(nrow):
        if (predList[irow] < 0.0) and (yOut[irow] >= 0.0):
            errCnt += 1.0
        elif (predList[irow] >= 0.0) and (yOut[irow] < 0.0):
            errCnt += 1.0
    misClassRate.append(errCnt/nrow)
```

```
#find minimum point for plot and for print
minError = min(misClassRate)
idxMin = misClassRate.index(minError)
plotAlphas = list(alphas[1:len(alphas)])

plot.figure()
plot.plot(plotAlphas, misClassRate,
    label='Misclassification Error Across Folds', linewidth=2)
plot.axvline(plotAlphas[idxMin], linestyle='--',
            label='CV Estimate of Best alpha')
plot.legend()
plot.semilogx()
ax = plot.gca()
ax.invert_xaxis()
plot.xlabel('alpha')
plot.ylabel('Misclassification Error')
plot.axis('tight')
plot.show()



#calculate AUC.
idxPos = [i for i in range(nrow) if yOut[i] > 0.0]
yOutBin = [0] * nrow
for i in idxPos: yOutBin[i] = 1

auc = []
for iPred in range(1, nPred):
    predList = list(pred[:, iPred])
    aucCalc = roc_auc_score(yOutBin, predList)
    auc.append(aucCalc)

maxAUC = max(auc)
idxMax = auc.index(maxAUC)

plot.figure()
plot.plot(plotAlphas, auc, label='AUC Across Folds', linewidth=2)
plot.axvline(plotAlphas[idxMax], linestyle='--',
            label='CV Estimate of Best alpha')
plot.legend()
plot.semilogx()
ax = plot.gca()
ax.invert_xaxis()
plot.xlabel('alpha')
plot.ylabel('Area Under the ROC Curve')
plot.axis('tight')
plot.show()


#plot best version of ROC curve
fpr, tpr, thresh = roc_curve(yOutBin, list(pred[:, idxMax]))
```

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()

print('Best Value of Misclassification Error = ', misClassRate[idxMin])
print('Best alpha for Misclassification Error = ', plotAlphas[idxMin])
print('')
print('Best Value for AUC = ', auc[idxMax])
print('Best alpha for AUC   =  ', plotAlphas[idxMax])

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

#pick some points along the curve to print. There are 208 points.
#The extremes aren't useful

#Sample at 52, 104 and 156. Use the calculated values of tpr and fpr
#along with definitions and threshold values.

#Some nomenclature (e.g. see wikipedia "receiver operating curve")


#P = Positive cases
P = len(idxPos)
#N = Negative cases
N = nrow - P
#TP = True positives = tpr * P
TP = tpr[52] * P
#FN = False negatives = P - TP
FN = P - TP
#FP = False positives = fpr * N
FP = fpr[52] * N
#TN = True negatives = N - FP
TN = N - FP

print('Threshold Value =    ', thresh[52])
print('TP = ', TP, 'FP = ', FP)
print('FN = ', FN, 'TN = ', TN)

TP = tpr[104] * P; FN = P - TP; FP = fpr[104] * N; TN = N - FP

print('Threshold Value =    ', thresh[104])
print('TP = ', TP, 'FP = ', FP)
print('FN = ', FN, 'TN = ', TN)

TP = tpr[156] * P; FN = P - TP; FP = fpr[156] * N; TN = N - FP

print('Threshold Value =    ', thresh[156])
print('TP = ', TP, 'FP = ', FP)
print('FN = ', FN, 'TN = ', TN)


Printed Output: [filename - rocksVMinesENetRegCVPrintedOutput.txt]
('Best Value of Misclassification Error = ', 0.22115384615384615)
('Best alpha for Misclassification Error = ', 0.017686244720179375)

('Best Value for AUC = ', 0.86867279650784812)
('Best alpha for AUC   =  ', 0.020334883589342503)

Confusion Matrices for Different Threshold Values
('Threshold Value =   ', 0.37952298245219962)
('TP = ', 48.0, 'FP = ', 5.0)
('FN = ', 63.0, 'TN = ', 92.0)
('Threshold Value =   ', -0.045503481125357965)
('TP = ', 85.0, 'FP = ', 20.0)
('FN = ', 26.0, 'TN = ', 77.0)
('Threshold Value =   ', -0.4272522354395466)
('TP = ', 107.0, 'FP = ', 49.999999999999993)
('FN = ', 4.0, 'TN = ', 47.000000000000007)
```

Cross-validation gives you a solid estimate of the performance that you are going to see when you deploy this system. If the performance indicated by cross-validation is not good enough, you will have to work to improve it. For example, you might try the basis expansion that was used in the section "Multivariable Regression: Predicting Wine Taste." You might also have a look at the cases giving the worst errors and see if you can discern a pattern, whether they're data-entry errors or if another variable can be added that would account for their being mistaken so badly. If the error satisfies the needs of your problem, you'll want to train a model on the whole data set for deployment. The next section runs through that process.

Build a Rocks versus Mines Classifier for Deployment

As with the wine quality case study, the next step is to retrain the model on the full data set and pull out the coefficients corresponding to the best alpha—the one determined to minimize out-of-sample error, which is estimated in this case study by cross-validation. Listing 5-5 shows the code for accomplishing this.

Listing 5-5: Coefficient Trajectories for ElasticNet Trained on Rocks versus Mines Data—rocksVMinesCoefCurves.py

```
__author__ = 'mike_bowles'
import urllib2
from math import sqrt, fabs, exp
import matplotlib.pyplot as plot
```

continues

continued

```
from sklearn.linear_model import enet_pathsh
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 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(1.0)
    else:
        labels.append(0.0)
    attrRow = [float(elt) for elt in row]
    xNum.append(attrRow)

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

alpha = 1.0

#calculate means and variances
xMeans = []
xSD = []
for i in range(ncol):
    col = [xNum[j][i] for j in range(nrow)]
    mean = sum(col)/nrow
    xMeans.append(mean)
    colDiff = [(xNum[j][i] - mean) for j in range(nrow)]
    sumSq = sum([colDiff[i] * colDiff[i] for i in range(nrow)])
    stdDev = sqrt(sumSq/nrow)
    xSD.append(stdDev)
```

```
#use calculate mean and standard deviation to normalize xNum
xNormalized = []
for i in range(nrow):
    rowNormalized = [(xNum[i][j] - xMeans[j])/xSD[j] \
        for j in range(ncol)]
    xNormalized.append(rowNormalized)

#normalize labels to center

meanLabel = sum(labels)/nrow
sdLabel = sqrt(sum([(labels[i] - meanLabel) * (labels[i] - meanLabel) \
        for i in range(nrow)])/nrow)

labelNormalized = [(labels[i] - meanLabel)/sdLabel for i in range(nrow)]

#Convert normalized labels to numpy array
Y = numpy.array(labelNormalized)

#Convert normalized attributes to numpy array
X = numpy.array(xNormalized)

alphas, coefs, _ = enet_path(X, Y,l1_ratio=0.8, fit_intercept=False,
    return_models=False)

plot.plot(alphas,coefs.T)

plot.xlabel('alpha')
plot.ylabel('Coefficients')
plot.axis('tight')
plot.semilogx()
ax = plot.gca()
ax.invert_xaxis()
plot.show()

nattr, nalpha = coefs.shape

#find coefficient ordering
nzList = []
for iAlpha in range(1,nalpha):
    coefList = list(coefs[: ,iAlpha])
    nzCoef = [index for index in range(nattr) if coefList[index] != 0.0]
    for q in nzCoef:
        if not(q in nzList):
            nzList.append(q)

#make up names for columns of X
names = ['V' + str(i) for i in range(ncol)]
nameList = [names[nzList[i]] for i in range(len(nzList))]
print("Attributes Ordered by How Early They Enter the Model")
```

continues

continued

```
print(nameList)
print('')
#find coefficients corresponding to best alpha value. alpha value
corresponding to normalized X and normalized Y is 0.020334883589342503

alphaStar = 0.020334883589342503
indexLTalphaStar = [index for index in range(100) if \
    alphas[index] > alphaStar]
indexStar = max(indexLTalphaStar)

#here's the set of coefficients to deploy
coefStar = list(coefs[:,indexStar])
print("Best Coefficient Values ")
print(coefStar)
print('')
#The coefficients on normalized attributes give another slightly
#different ordering

absCoef = [abs(a) for a in coefStar]

#sort by magnitude
coefSorted = sorted(absCoef, reverse=True)

idxCoefSize = [absCoef.index(a) for a in coefSorted if not(a == 0.0)]

namesList2 = [names[idxCoefSize[i]] for i in range(len(idxCoefSize))]

print("Attributes Ordered by Coef Size at Optimum alpha")
print(namesList2)

Printed Output: [filename - rocksVMinesCoefCurvesPrintedOutput.txt]
Attributes Ordered by How Early They Enter the Model
['V10', 'V48', 'V11', 'V44', 'V35', 'V51', 'V20', 'V3', 'V21', 'V45',
'V43', 'V15', 'V0', 'V22', 'V27', 'V50', 'V53', 'V30', 'V58', 'V56',
'V28', 'V39', 'V46', 'V19', 'V54', 'V29', 'V57', 'V6', 'V8', 'V7',
'V49', 'V2', 'V23', 'V37', 'V55', 'V4', 'V13', 'V36', 'V38', 'V26',
'V31', 'V1', 'V34', 'V33', 'V24', 'V16', 'V17', 'V5', 'V52', 'V41',
'V40', 'V59', 'V12', 'V9', 'V18', 'V14', 'V47', 'V42']

Best Coefficient Values
[0.082258256813766639, 0.0020619887220043702, -0.11828642590855878,
0.16633956932499627, 0.0042854388193718004, -0.0, -0.043662524745940004,
 -0.07751510487942842, 0.10000054356323497, 0.0, 0.090617207036282038,
0.21210870399915693, -0.0, -0.010655386149821946, -0.0,
-0.13328659558143779, -0.0, 0.0, 0.0, 0.052814854501417867,
0.038531154796719078, 0.0035515348181877982, 0.090854714680378215,
0.030316113904025031, -0.0, 0.0, 0.0086195542357481014, 0.0, 0.0,
0.17497679257272536, -0.2215687804617206, 0.012614243827937584,
0.0, -0.0, 0.0, -0.17160601809439849, -0.080450013824209077,
0.078096790041518344, 0.022035287616766441, -0.072184409273692227,
0.0, -0.0, 0.0, 0.057018816876250704, 0.096478265685721556,
0.039917367637236176, 0.049158231541622875, 0.0, 0.22671917920123755,
-0.096272735479951091, 0.0, 0.078886784332226484, 0.0,
0.062312821755756878, -0.082785510713295471, 0.014466967172068596,
-0.074326527525632721, 0.068096475974257331,
0.070488864435477847, 0.0]

Attributes Ordered by Coef Size at Optimum alpha
['V48', 'V30', 'V11', 'V29', 'V35', 'V3', 'V15', 'V2', 'V8', 'V44',
'V49', 'V22', 'V10', 'V54', 'V0', 'V36', 'V51', 'V37', 'V7', 'V56',
'V39', 'V58', 'V57', 'V53', 'V43', 'V19', 'V46', 'V6', 'V45', 'V20',
'V23', 'V38', 'V55', 'V31', 'V13', 'V26', 'V4', 'V21', 'V1']
```

The code in Listing 5-5 is structured similarly to the code in Listing 5-4, except that there's no cross-validation loop. The value for alpha at which coefficients are sought is hard-coded and comes directly from the results generated by Listing 5-4. There were two values of alpha generated: one that minimized the misclassification error and one that maximized the AUC. The alpha that maximized AUC was slightly larger and slightly more conservative. It was slightly to the left of the value that minimized misclassification error and therefore slightly more conservative. The coefficients printed by the program are listed at the bottom of the code. Out of the 60 coefficients, 20-some-odd are 0. In this run (as in the cross-validation program), the `l1_ratio` variable was set to `0.8`, which typically results in more coefficients than Lasso regression, which would correspond to `l1_ratio` at `1.0`.

A couple of measures of variable importance are printed at the bottom of the listing. One is the order in which variables come into the solution as alpha is decremented downward. The other ordering is according to the magnitude of the coefficients at the optimum solution. As discussed in conjunction with the wine quality data, these orderings only make sense when the attributes are normalized. Some degree of agreement exists between these two different variable orderings, but they don't agree completely. For example, the variables V48, V11, V35, V44, and V3 appear relatively high in both lists, but V10 appears at the top of the first list and is much further down in the ordering based on coefficient size. Apparently, V10 is important when the coefficient penalty is so large that the algorithm only permits a single attribute, but when the coefficient penalty has shrunk to the point that a multitude of attributes are included, the attribute V10 levels off and drops in importance somewhat as other attributes are added to the mix.

Typically, objects give the strongest reflections for waves whose wavelength is the same order of characteristic dimensions of the object. Mines (metal cylinders) have length and diameter—relatively few and relatively long characteristic dimensions to reflect compared to rocks, which are more fractal in character and reflect a broader range of wavelengths. Because all the attribute values in the data set are positive (power levels), you might expect that the wavelengths corresponding to low frequencies would get positive coefficients and the wavelengths corresponding to high frequencies would get negative coefficients.

You can see how this differencing could easily lead to overfitting the data and building a model that did extremely well on this data set but didn't generalize. The cross-validation process ensures that the model isn't overfit as long as the training data is statistically similar to what the model will see in deployment. The errors seen in cross-validation will match those in deployment to the extent that the rocks and mines encountered in deployment match the nature and proportions of those in the training data.

Figure 5-10 plots the coefficient curves for the ElasticNet regression models trained on the full rocks versus mines data set. The curves emphasize the complexity and changing nature of the relative importance of the available attributes.

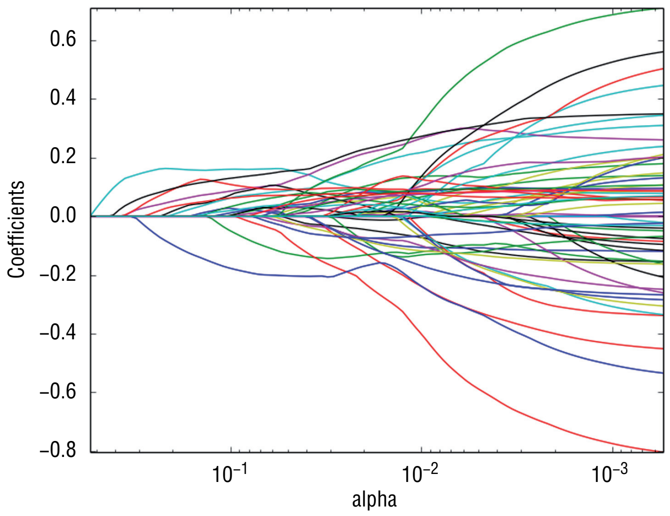

Figure 5-10: Coefficient curves for ElasticNet trained on rocks versus mines data

As mentioned in Chapter 4, an alternative to using penalized regression for classification is to use penalized logistic regression. Listing 5-5 shows code for an implementation of penalized logistic regression to build a classifier for the rocks versus mines data. The listing and the associated results highlight the similarities and differences between the two approaches. The algorithmic differences can be seen in the structure of the iteration. The logistic regression approach involves using linear functions of the attributes to calculate probabilities and likelihoods of each of the training examples being a rock or a mine. (See `http://en.wikipedia.org/wiki/Logistic_regression` for more background on logistic regression and for careful derivations of the associated equations.) The algorithm for nonpenalized logistic regression is called *iteratively reweighted least squares* (IRLS). The name comes from the nature of the algorithm (see `http://en.wikipedia.org/wiki/Iteratively_reweighted_least_squares`). It derives weights based on probability estimates for each example in the training set. Given the weights, the problem becomes a weighted least squares regression problem. The process has to be iterated until the probabilities (and corresponding weights) stop changing. Basically, the IRLS for logistic regression adds another layer of iteration to the algorithm for (not logistic) penalized regression you saw in Chapter 4.

After reading in the variables and normalizing them, the program initializes weights and probabilities that are central to logistic regression and to the penalized version of it. These probabilities and weights have to be estimated along with the coefficients (β's) each time the penalty parameter decrements. You'll see the letters IRLS added to some of the variables in the code to denote that they are associated with the IRLS layer of the iteration. The iteration to estimate the probabilities is inside the loop for decrementing the λ's and wraps around the loop for iterating the coordinate descent on the β's.

The details of the update are slightly more complicated than the algorithm for plain (not logistic) penalized regression. One complication is the weights that come with IRLS. The weights and probabilities get calculated one input example at a time. Those are denoted by `p` and `w` in the code. The effects of the weights on sums of products like *attributes times residuals* and *squares of attributes* also need to be collected. Those are denoted by variables like `sumWxx`, which is a list containing the sum of the weights times each of the attributes squared. The other complication is that the residuals are now a function of the labels, the probabilities, and (more familiarly) the attributes and their coefficients (β's).

The code runs and produces variable ordering and coefficient curves to compare with those generated using nonlogistic penalized regression. The logistic transformation makes direct comparison of the coefficients problematic because the logistic function causes a nonlinear scale change. Both plain and logistic regression (penalized and nonpenalized) generate vectors of coefficients and then multiply the (same) attributes by them and compare to a threshold. The threshold value is somewhat secondary since it can be determined subsequent to training, as was demonstrated. So the overall scale of the β's doesn't matter as much as the magnitudes of the components relative to one another. One way to judge the relative magnitudes is to look at the order in which the two methods bring in new variables. As you can see by comparing the printed output in Listing 5-5 to the printed output in Listing 5-4, the two methods agree completely on the ordering for the first eight attributes. Of the next eight variables, seven of the eight are common to both lists, although they are ordered somewhat differently. Roughly the same is true of the next eight. There's fairly good general agreement in the ordering between the two methods.

Another question is which one delivers better performance. Assessing that requires running cross-validation with penalized logistic regression. You have the tools and code to carry that out. The code in Listing 5-6 is not at all optimized for speed, but it won't take too long on the rocks versus mines problem.

Listing 5-6: Penalized Logistic Regression Trained on Rocks versus Mines Data— rocksVMinesGlmnet.py

```
__author__ = 'mike_bowles'
import urllib2
import sys
from math import sqrt, fabs, exp
import matplotlib.pyplot as plot

def S(z,gamma):
    if gamma >= fabs(z):
        return 0.0
    if z > 0.0:
        return z - gamma
    else:
        return z + gamma

def Pr(b0,b,x):
    n = len(x)
    sum = b0
    for i in range(n):
        sum += b[i]*x[i]
        if sum < -100: sum = -100
    return 1.0/(1.0 + exp(-sum))


#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(1.0)
    else:
        labels.append(0.0)
    attrRow = [float(elt) for elt in row]
    xNum.append(attrRow)

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

alpha = 0.8
#calculate means and variances
xMeans = []
xSD = []
for i in range(ncol):
    col = [xNum[j][i] for j in range(nrow)]
    mean = sum(col)/nrow
    xMeans.append(mean)
    colDiff = [(xNum[j][i] - mean) for j in range(nrow)]
    sumSq = sum([colDiff[i] * colDiff[i] for i in range(nrow)])
    stdDev = sqrt(sumSq/nrow)
    xSD.append(stdDev)

#use calculate mean and standard deviation to normalize xNum
xNormalized = []
for i in range(nrow):
    rowNormalized = [(xNum[i][j] - xMeans[j])/xSD[j] \
        for j in range(ncol)]
    xNormalized.append(rowNormalized)

#Do Not Normalize labels but do calculate averages
meanLabel = sum(labels)/nrow
sdLabel = sqrt(sum([(labels[i] - meanLabel) * (labels[i] - meanLabel) \
    for i in range(nrow)])/nrow)

#initialize probabilities and weights
sumWxr = [0.0] * ncol
sumWxx = [0.0] * ncol
sumWr = 0.0
sumW = 0.0

#calculate starting points for betas
for iRow in range(nrow):
    p = meanLabel
    w = p * (1.0 - p)
    #residual for logistic
    r = (labels[iRow] - p) / w
    x = xNormalized[iRow]
    sumWxr = [sumWxr[i] + w * x[i] * r for i in range(ncol)]
    sumWxx = [sumWxx[i] + w * x[i] * x[i] for i in range(ncol)]
    sumWr = sumWr + w * r
    sumW = sumW + w
```

continues

continued

```
avgWxr = [sumWxr[i]/nrow for i in range(ncol)]
avgWxx = [sumWxx[i]/nrow for i in range(ncol)]

maxWxr = 0.0
for i in range(ncol):
    val = abs(avgWxr[i])
    if val > maxWxr:
        maxWxr = val

#calculate starting value for lambda
lam = maxWxr/alpha

#this value of lambda corresponds to beta = list of 0's
#initialize a vector of coefficients beta
beta = [0.0] * ncol
beta0 = sumWr/sumW

#initialize matrix of betas at each step
betaMat = []
betaMat.append(list(beta))

beta0List = []
beta0List.append(beta0)

#begin iteration
nSteps = 100
lamMult = 0.93 #100 steps gives reduction by factor of 1000 in lambda
               #(recommended by authors)
nzList = []
for iStep in range(nSteps):
    #decrease lambda
    lam = lam * lamMult


    #Use incremental change in betas to control inner iteration


    #set middle loop values for betas = to outer values
    #values are used for calculating weights and probabilities
    #inner values are used for calculating penalized regression updates

    #take pass through data to calculate averages over data required
    #for iteration
    #initilize accumulators

    betaIRLS = list(beta)
    beta0IRLS = beta0
    distIRLS = 100.0
    #Middle loop to calculate new betas with fixed IRLS weights and
    #probabilities
    iterIRLS = 0
```

```
while distIRLS > 0.01:
    iterIRLS += 1
    iterInner = 0.0

    betaInner = list(betaIRLS)
    beta0Inner = beta0IRLS
    distInner = 100.0
    while distInner > 0.01:
        iterInner += 1
        if iterInner > 100: break

        #cycle through attributes and update one-at-a-time
        #record starting value for comparison
        betaStart = list(betaInner)
        for iCol in range(ncol):

            sumWxr = 0.0
            sumWxx = 0.0
            sumWr = 0.0
            sumW = 0.0

            for iRow in range(nrow):
                x = list(xNormalized[iRow])
                y = labels[iRow]
                p = Pr(beta0IRLS, betaIRLS, x)
                if abs(p) < 1e-5:
                    p = 0.0
                    w = 1e-5
                elif abs(1.0 - p) < 1e-5:
                    p = 1.0
                    w = 1e-5
                else:
                    w = p * (1.0 - p)

                z = (y - p) / w + beta0IRLS + sum([x[i] *
                    betaIRLS[i] for i in range(ncol)])
                r = z - beta0Inner - sum([x[i] * betaInner[i]
                    for i in range(ncol)])
                sumWxr += w * x[iCol] * r
                sumWxx += w * x[iCol] * x[iCol]
                sumWr += w * r
                sumW += w

            avgWxr = sumWxr / nrow
            avgWxx = sumWxx / nrow

            beta0Inner = beta0Inner + sumWr / sumW
            uncBeta = avgWxr + avgWxx * betaInner[iCol]
            betaInner[iCol] = S(uncBeta, lam * alpha) / (avgWxx +
                lam * (1.0 - alpha))
```

continues

continued

```
            sumDiff = sum([abs(betaInner[n] - betaStart[n]) \
                for n in range(ncol)])
            sumBeta = sum([abs(betaInner[n]) for n in range(ncol)])
            distInner = sumDiff/sumBeta
        #print number of steps for inner and middle loop convergence
        #to monitor behavior
        #print(iStep, iterIRLS, iterInner)

        #if exit inner while loop, then set betaMiddle = betaMiddle
        #and run through middle loop again.

        #Check change in betaMiddle to see if IRLS is converged
        a = sum([abs(betaIRLS[i] - betaInner[i]) for i in range(ncol)])
        b = sum([abs(betaIRLS[i]) for i in range(ncol)])
        distIRLS = a / (b + 0.0001)
        dBeta = [betaInner[i] - betaIRLS[i] for i in range(ncol)]
        gradStep = 1.0
        temp = [betaIRLS[i] + gradStep * dBeta[i] for i in range(ncol)]
        betaIRLS = list(temp)

    beta = list(betaIRLS)
    beta0 = beta0IRLS
    betaMat.append(list(beta))
    beta0List.append(beta0)

    nzBeta = [index for index in range(ncol) if beta[index] != 0.0]
    for q in nzBeta:
        if not(q in nzList):
            nzList.append(q)

#make up names for columns of xNum
names = ['V' + str(i) for i in range(ncol)]
nameList = [names[nzList[i]] for i in range(len(nzList))]

print("Attributes Ordered by How Early They Enter the Model")
print(nameList)
for i in range(ncol):
    #plot range of beta values for each attribute
    coefCurve = [betaMat[k][i] for k in range(nSteps)]
    xaxis = range(nSteps)
    plot.plot(xaxis, coefCurve)

plot.xlabel("Steps Taken")
plot.ylabel("Coefficient Values")
plot.show()

Printed Output: [filename - rocksVMinesGlmnetPrintedOutput.txt]

Attributes Ordered by How Early They Enter the Model
['V10', 'V48', 'V11', 'V44', 'V35', 'V51', 'V20', 'V3', 'V50', 'V21',
'V43', 'V47', 'V15', 'V27', 'V0', 'V22', 'V36', 'V30', 'V53', 'V56',
'V58', 'V6', 'V19', 'V28', 'V39', 'V49', 'V7', 'V23', 'V54', 'V8',
'V14', 'V2', 'V29', 'V38', 'V57', 'V45', 'V13', 'V32', 'V31', 'V42',
 'V16', 'V37', 'V59', 'V52', 'V25', 'V18', 'V1', 'V33', 'V4', 'V55',
'V17', 'V46', 'V26', 'V12', 'V40', 'V34', 'V5', 'V24', 'V41', 'V9']
```

Figure 5-11 shows the coefficient curves for rocks versus mines using penalized logistic regression. As noted, the scale of the coefficients is different from plain penalized regression because of the logistic function difference between the two methods. Ordinary regression attempts to fit a straight line to targets that are 0.0 and 1.0. Logistic regression attempts to predict probabilities of class membership by fitting a straight line to the "log odds ratio." Suppose p is the predicted probability that an example corresponds to the mines class. Then the odds ratio is the ratio $\frac{p}{1-p}$. The log odds ratio is the natural log of the odds ratio. Whereas p ranges from 0 to 1, the log odds ratio of p ranges from minus infinity to plus infinity. The cases where the log odd is very large and positive corresponds to cases where the prediction is very certain that the case belongs to the mines class. Ones that are large negative numbers correspond to the rocks class.

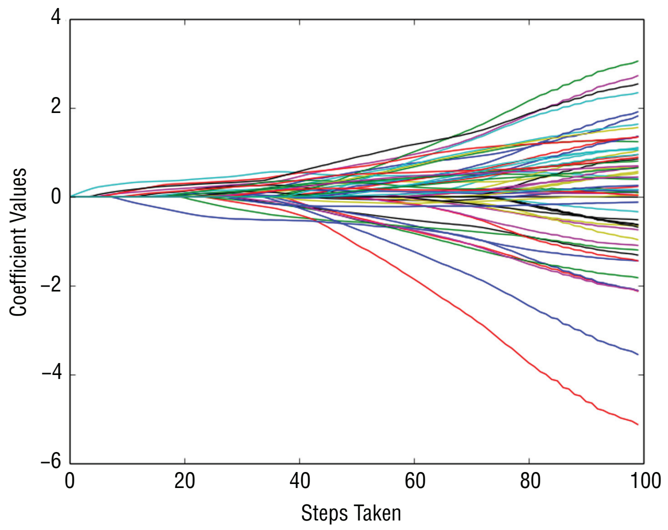

Figure 5-11: Coefficient curves for ElasticNet penalized logistic regression trained on rocks versus mines data

Because the two methods are predicting vastly different quantities, the scale on the predictions is much different and the coefficients are correspondingly different. But as the printed output from the two programs indicates, the order in which the variables appear in the solution is very similar, and the coefficient curves show that the signs are the same for the first several attributes that enter the solution.

Multiclass Classification: Classifying Crime Scene Glass Samples

The rocks versus mines problem that you saw in the last section is called a binary classification problem because the labels and predictions take one of two possible values. (Did the sonar return being processed come from reflections off a rock or a mine?) If labels and predictions can take more than two values, the problem is called a multiclass classification problem. This section uses penalized linear regression for the problem of classifying glass samples. As described more fully in Chapter 2, the glass data set consists of 9 physical chemistry measurements (refractive index and measurements of chemical composition) on 214 samples of 6 different types of glass. The problem is to use the physical chemistry measurements to determine which of the six types a given sample represents. The application for this is forensic analysis of crime and accident scenes. The data set comes from the UCI data repository, and the web page for the data set references a paper that uses support vector machines to solve this same problem. After looking at the code for solving this problem, this section will compare performance with the support vector machine approach.

Listing 5-7 shows code for solving this problem.

Listing 5-7: Multiclass Classification with Penalized Linear Regression - Classifying Crime Scene Glass Samples—glassENetRegCV.py

```
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
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)

names = ['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
nrow = len(xNum)
ncol = len(xNum[1])

#create one versus all label vectors
#get distinct glass types and assign index to each
yOneVAll = []
labelSet = set(labels)
labelList = list(labelSet)
labelList.sort()
nlabels = len(labelList)
for i in range(nrow):
    yRow = [0.0]*nlabels
    index = labelList.index(labels[i])
    yRow[index] = 1.0
    yOneVAll.append(yRow)

#calculate means and variances
xMeans = []
xSD = []
for i in range(ncol):
    col = [xNum[j][i] for j in range(nrow)]
    mean = sum(col)/nrow
    xMeans.append(mean)
    colDiff = [(xNum[j][i] - mean) for j in range(nrow)]
    sumSq = sum([colDiff[i] * colDiff[i] for i in range(nrow)])
    stdDev = sqrt(sumSq/nrow)
    xSD.append(stdDev)

#use calculate mean and standard deviation to normalize xNum
xNormalized = []
for i in range(nrow):
    rowNormalized = [(xNum[i][j] - xMeans[j])/xSD[j] \
        for j in range(ncol)]
    xNormalized.append(rowNormalized)

#normalize y's to center
yMeans = []
ySD = []
for i in range(nlabels):
    col = [yOneVAll[j][i] for j in range(nrow)]
```

continues

continued

```
    mean = sum(col)/nrow
    yMeans.append(mean)
    colDiff = [(yOneVAll[j][i] - mean) for j in range(nrow)]
    sumSq = sum([colDiff[i] * colDiff[i] for i in range(nrow)])
    stdDev = sqrt(sumSq/nrow)
    ySD.append(stdDev)

yNormalized = []
for i in range(nrow):
    rowNormalized = [(yOneVAll[i][j] - yMeans[j])/ySD[j] \
        for j in range(nlabels)]
    yNormalized.append(rowNormalized)


#number of cross-validation folds
nxval = 10
nAlphas=200
misClass = [0.0] * nAlphas

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

    #Define test and training attribute and label sets
    xTrain = numpy.array([xNormalized[r] for r in idxTrain])
    xTest = numpy.array([xNormalized[r] for r in idxTest])
    yTrain = [yNormalized[r] for r in idxTrain]
    yTest = [yNormalized[r] for r in idxTest]
    labelsTest = [labels[r] for r in idxTest]

    #build model for each column in yTrain
    models = []
    lenTrain = len(yTrain)
    lenTest = nrow - lenTrain
    for iModel in range(nlabels):
        yTemp = numpy.array([yTrain[j][iModel]
            for j in range(lenTrain)])
        models.append(enet_path(xTrain, yTemp,l1_ratio=1.0,
            fit_intercept=False, eps=0.5e-3, n_alphas=nAlphas ,
            return_models=False))

    for iStep in range(1,nAlphas):
        #Assemble the predictions for all the models, find largest
        #prediction and calc error
        allPredictions = []
        for iModel in range(nlabels):
            _, coefs, _ = models[iModel]
            predTemp = list(numpy.dot(xTest, coefs[:,iStep]))
            #un-normalize the prediction for comparison
            predUnNorm = [(predTemp[j]*ySD[iModel] + yMeans[iModel]) \
                for j in range(len(predTemp))]
            allPredictions.append(predUnNorm)

        predictions = []
        for i in range(lenTest):
            listOfPredictions = [allPredictions[j][i] \
                for j in range(nlabels) ]
            idxMax = listOfPredictions.index(max(listOfPredictions))
            if labelList[idxMax] != labelsTest[i]:
                misClass[iStep] += 1.0

misClassPlot = [misClass[i]/nrow for i in range(1, nAlphas)]

plot.plot(misClassPlot)

plot.xlabel("Penalty Parameter Steps")
plot.ylabel(("Misclassification Error Rate"))
plot.show()
```

The first part of the code deals with reading the data from the UCI website and separating the labels from the attributes. The attributes get normalized in the usual way. The one-versus-all approach leads to some distinctive changes in the treatment of the labels. Instead of having a single set of labels, the one-versus-all approach leads to as many vectors of labels as there are distinct labels. In the glass problem, there are six different labels. So, where the regression and binary classification problems had a single vector of labels, the glass problem has six vectors of labels. The intuition behind this is as follows: If you've got a problem of dividing a set of points into two groups, one plane will do it. If your problem is to divide a set of points into six groups, you'll need more than a single plane.

One versus all trains as many different binary classifiers as there are distinct labels. The difference between these different classifiers is that they are trained to different labels. The code in Listing 5-7 shows how these labels are constructed from the original multiclass labels given by the problem. The approach is very similar to the approach you saw in Chapter 4 for converting categorical variables to numeric variables. The code listing extracts the distinct labels using a Python set, orders them from smallest to largest (not really necessary but helpful for keeping things straight), and then forms a column of labels where the first column has a 1 if the original labels take the first distinct label and has a 0 otherwise, the second column has a 1 if the original label takes the second distinct label, and so on. You can see why this is called one versus all. The labels in the first column will lead to a binary classifier predicting whether the sample takes the value of the first distinct label. Each of the six classifiers has a similar binary decision to make.

The code goes on to build a cross-validation loop along familiar lines. One minor difference is that the raw labels are also sliced into a test set to facilitate measuring misclassification error later on. There's a signal difference in the model training because on each cross-validation fold six models are trained and the trained models are stored in a list for later use. There are a couple of changes to the call to `enet_path` that are useful to discuss. One is that the `eps` parameter is spelled out and is exactly half the default value of `1e-3`. This is one of the parameters that you have at your disposal to control the range of penalty parameter values that are covered in the training. Recall from the discussion (and code example) in Chapter 4 that the coordinate descent algorithm progresses by decrementing the penalty parameter. The `eps` parameter tells the algorithm where to stop decrementing. The input `eps` is the ratio of the stopping value of the penalty parameter divided by the starting value. The parameter `n_alphas` controls the number of steps. Be aware that taking steps that are too large my result in the algorithm not converging. It will give you a warning message if it fails to converge. You can then either make `eps` a little larger so the penalty parameter doesn't get decremented by quite so much each step or you can take more steps by making `n_alphas` larger and thereby making each individual step smaller.

Another factor to consider is whether you're seeing enough of the curve. The plot in Figure 5-12 shows a minimum that's fairly close to the right edge of the graph. It would be useful to see a little more of the curve to be sure that the minimum isn't further to the right. Decreasing `eps` will show portions of the curve to the right of where the curve currently ends.

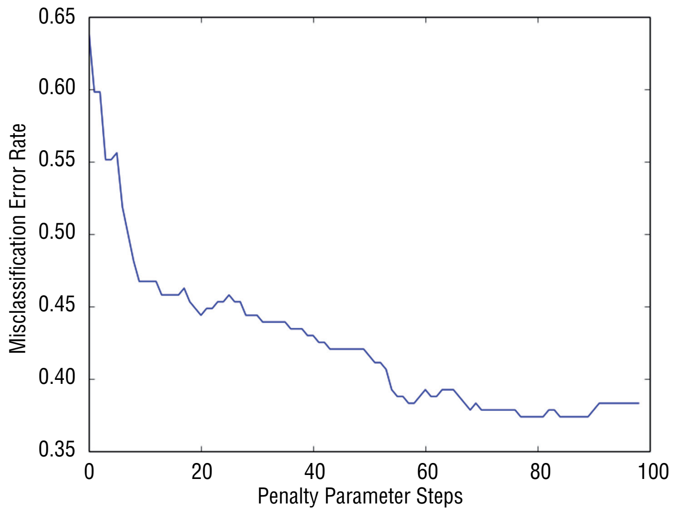

Figure 5-12: Misclassification error rates using penalized linear regression for glass classification

After the six models are trained, they are used to make six predictions. The code checks to see which of the six predictions has the largest numerical value and chooses the corresponding value for the prediction. Then that is compared to the actual value and the error is accumulated.

Figure 5-12 shows a plot of the misclassification error rate versus the number of decrementing steps that the penalty parameter has undergone. The plot shows a marked improvement at the minimum from the simplest model at the left-hand edge of the plot. The minimum value for misclassification error is roughly 35 percent. This is a better value than reported for a linear kernel support vector machine. That paper does achieve misclassification errors of 35 percent for some choices of nonlinear kernels and gets errors as low as 30 percent for some nonlinear kernels. Using nonlinear kernels in a support vector machine is roughly equivalent to basis expansion that you saw used in the wine quality example earlier in this chapter. Basis expansion didn't prove effective in the wine quality problem, but the fact that nonlinear kernels gave marked performance improvement for support vector machines makes that a promising method for improving performance in the glass classification problem.

Summary

This chapter demonstrated the use of penalized regression along with a number of general tools for predictive modeling. The chapter showed several different types of problems that you'll frequently encounter in real problems. These include regression, binary classification, and multiclass classification. The chapter used Python packages incarnating various different flavors of penalized regression for these tasks. In addition, the chapter illustrated the use of several tools that you may need in order to solve the modeling problems that you encounter. These include techniques for coding factor variables as numeric, for using a binary classifier to solve multiclass classification problems, and for extending linear methods to predict nonlinear relationships between attributes and outcomes.

The chapter also demonstrated a variety of ways to quantify performance for your predictive models. Regression problems are easiest to quantify because their errors can naturally be expressed in real number terms. Classification problems can be more involved. You saw classification performance quantified as misclassification error rates, area under the receiver operating curve, and economic costs. You should pick the method that comes closest to measuring performance in terms of your actual objectives (business objectives, science objectives, and so forth).

References

1. P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis. (2009). Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, Elsevier, 47(4):547–553.
2. T. Hastie, R. Tibshirani, and J. Friedman. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* 2nd ed. Springer-Verlag, New York.
3. J. Friedman, T. Hastie, and R. Tibshirani. (2010). Regularization paths for generalized linear models via coordinate descent. *Journal of Statistical Software*, 33(1).
4. K. Bache and M. Lichman. (2013). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science. `http://archive.ics.uci.edu/ml`.