CHAPTER

4

Penalized Linear Regression

As you saw in Chapter 3, "Predictive Model Building: Balancing Performance, Complexity, and Big Data," getting linear regression to work in practice requires some manipulation of the ordinary least squares algorithm. Ordinary least squares regression cannot temper its use of all the data available in an attempt to minimize the error on the training data. Chapter 3 illustrated that this situation can lead to models that perform much worse on new data than on the training data. Chapter 3 showed two extensions of ordinary least squares regression. Both of these involved judiciously reducing the amount of data available to ordinary least squares and using out-of-sample error measurement to determine how much data resulted in the best performance.

Stepwise regression began by letting ordinary least squares regression use exactly one of the attribute columns for making predictions and by picking the best one. It proceeded by adding new attributes to the existing model.

Ridge regression introduced a different type of constraint. Ridge regression imposed a penalty on the magnitude of the coefficients to constrict the solution. Both ridge regression and forward stepwise regression gave better than ordinary least squares (OLS) on example problems.

This chapter develops an extended family of methods for taming the overfitting inherent in OLS. The methods discussed in this chapter are called *penalized linear regression*. Penalized linear regression covers several algorithms that operate similarly to the methods introduced in Chapter 3. Ridge regression is a specific example of a penalized linear regression algorithm. Ridge regression regulates overfitting by penalizing the sum of the regression coefficients squared. Other penalized regression algorithms use different forms of penalty. This chapter explains how the penalty method determines the nature of the solution and the type of information that is available about the solution.

Why Penalized Linear Regression Methods Are So Useful

Several properties make penalized linear regression methods outstandingly useful, including the following:

- Extremely fast model training
- Variable importance information
- Extremely fast evaluation when deployed
- Reliable performance on a wide variety of problems—particularly on attribute matrices that are not very tall compared to their width or that are sparse. Sparse solutions (that is, a more parsimonious model)
- May require linear model

Here's what these properties mean to you as a designer of machine learning models.

Extremely Fast Coefficient Estimation

Training time matters for several reasons. One reason is that the process of building a model is iterative. You'll find that you use training as part of your feature selection and feature engineering process. You'll pick some features that seem reasonable, train a model, evaluate it on out-of-sample data, want more performance, make some changes, and try again. If the basic training gets done quickly, you don't waste so much time getting coffee while waiting for answers (and reap the health benefit of lowering your caffeine intake). This makes the development process faster. Another reason why training times matter is that you might need to retrain your models to keep them working as conditions change. If you're classifying tweets, you might need to stay on top of changes in vocabulary. If you're training to trade in financial markets, the conditions are always changing. The time taken for training, even without feature reengineering, will dictate how rapidly you can respond to changing conditions.

Variable Importance Information

Both classes of algorithms covered in this book develop variable importance information. Variable importance information consists of a ranking for each of the attributes you've chosen to base your model on. The ranking tells you how much the model values each attribute compared to others. A highly ranked attribute contributes more to the model's prediction than lesser-ranked attributes. This is crucial information for a variety of reasons. First, it helps you weed out variables during the feature engineering process. The good features will rise to the top of the list, and the not-so-good ones will sink to the bottom. Besides helping you with feature engineering, knowing what variables are driving the predictions helps you understand and explain your models to others (your boss, your customer, subject matter experts in the company, and so on). To the extent that the important attributes are what people expected it gives them confidence that the models make sense. If some of the rankings are surprises, you may gain new insights into your problem. Discussion about the relative importance can give your development group new ideas about where to look for performance improvements.

The two properties of rapid training and variable importance make penalized regression a good algorithm to try first on a new problem. These algorithms help you quickly get your arms around the problems and decide which features are going to be useful.

Extremely Fast Evaluation When Deployed

In some problem settings, fast evaluations are a critical performance parameter. In some electronic markets (for example, Internet ads and automated trading), whoever gets the answer first gets the business. In many other applications (for instance, spam filtering), time might be critical, although not a yes/no criterion. It is hard to beat a linear model for evaluation speed. The number of operations required for the prediction calculation is one multiply and one add for each attribute.

Reliable Performance

Reliable performance means that penalized linear methods will generate reasonable answers to problems of all different shapes and sizes. On some problems, they will equal the best performance available. In some cases, they will outperform all contenders with a little coaxing. This chapter will talk about the sorts of coaxing available. Chapter 6, "Ensemble Methods," revisits this topic and explains some ways to use penalized linear regression in conjunction with ensemble methods to improve performance.

Sparse Solutions

A *sparse solution* means that many of the coefficients in the model are zero. That means that not as many multiplications and sums are required. More important, a sparse model (one with few nonzero coefficients) is easier to interpret. It's easier to see what attributes are driving the predictions that the model is generating.

Problem May Require Linear Model

The last reason for using penalized linear regression is that a linear model might be a requirement of the solution. Calculations of insurance payouts represent one example where linear models are required, where a payout formula is often part of a contract that specifies variables and their coefficients. An ensemble model that involves a thousand trees, each with a thousand parameters, would be nearly impossible to write out in English. Drug testing is another arena where regulatory apparatus requires a linear form for statistical inference.

When to Use Ensemble Methods

The prime reason for not using penalized linear regression is that you might get better performance with another technique, such as an ensemble method. As outlined in Chapter 3, ensembles perform best in complicated problems (for example, highly irregular decision surfaces) with plenty of data to resolve the problem's complexities. In addition, ensemble methods for measuring variable importance can yield more information about the relationship between attributes and predictions. For example, ensembles will give second-order (and higher) information about what pairs of variables are more important together than the sum of their individual importance. That information can actually help squeeze more performance out of penalized regression. You'll read more about that in Chapter 6.

Penalized Linear Regression: Regulating Linear Regression for Optimum Performance

As discussed in Chapter 3, this book addresses a class of problems called *function approximation*. The starting point for training a model for a function approximation problem is a data set containing a number of examples or instances. Each instance has an outcome (also called a *target*, *label*, *endpoint*, and so forth) and a number of attributes that are used to predict the outcome. Chapter 3 gave a simple illustrative example. It is repeated here in slightly modified form as Table 4-1.

Table 4-1: Example Training Set

OUTCOMES	FEATURE 1	FEATURE 2	FEATURE 3
$ Spent 2013	Gender	$ Spent 2012	Age
100	M	0.0	25
225	F	250	32
75	F	12	17

In this table, the outcomes are real-valued—making this a regression problem. The gender attribute (Feature 1) is two-valued, making it a categorical (or factor) attribute. The other two attributes are numeric. The goal with a function approximation problem is to (1) build a function relating the attributes to the outcome and (2) to minimize the error in some sense. Chapter 3 discussed some of the alternative error characterizations that might be employed to quantify overall error.

Data sets of the type shown in Table 4-1 are often represented by a column vector containing the outcomes (the leftmost column) and a matrix containing the attributes (the three columns of features). Asserting that the feature columns fit into a matrix abuses mathematical language a little. Strictly speaking, a matrix contains elements that are all defined over the same field. The contents of a matrix can all be real numbers, integers, complex numbers, binary numbers, and so on. They cannot, however, be a mixture of real numbers and categorical variables.

Here's an important point. Linear methods work with numeric data only. The data in Table 4-1 has non-numeric data, and therefore linear methods will not work for the data as shown. Fortunately, it is relatively simple to convert (or code) the data in Table 4-1 as numeric data. You'll learn the technique for coding categorical attributes as numeric attributes in the section titled "Incorporating Non-Numeric Attributes into Linear Methods." Given that the attributes are all real numbers (either in the initial problem formulation or by coding categorical attributes as real numbers), the data for a linear regression problem can be represented by two objects: Y and X, where Y is a column vector of outcomes, and where X is a matrix of real-valued attributes.

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Equation 4-1: Vector of outcomes

In the example given in Table 4-1, Y is the column labeled Outcomes.

$$X = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & & \ddots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{pmatrix}$$

Equation 4-2: Matrix of attributes

In the example given in Table 4-1, X is the set of columns that remains after excluding the Outcomes column.

The ith element from $Y(y_i)$ is from the same instance as the ith row of X. The ith row of X will be denoted by x_i with a single subscript and given by $x_i = (x_{i1}, x_{i2}, \quad , x_{im})$. The ordinary least squares regression problem is to minimize the error between the y_i and a linear function x_i, the ith row of attributes from X (that is, to find a vector of real numbers β).

$$\beta = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{pmatrix}$$

Equation 4-3: β - Vector of model coefficients

and a scalar β_0 so so that each element y_i from Y is approximated by

$$\begin{aligned} &\mathit{Prediction\,of}\ y_i = x_i * \beta + \beta_0 \\ &= x_{i1} * \beta_1 + x_{i2} * \beta_2 + \cdots + x_{im} * \beta_m + \beta_0 \end{aligned}$$

Equation 4-4: Linear relation between X and prediction of Y

You might be able to find the values for the β's by using your knowledge of the subject matter. In Table 4-1, for example, you might estimate that people will spend 10% more in 2013 than in 2012, that their purchases will increase by \$10 per year of age, and that even newborns will purchase \$50 of books. That gives you an equation to predict book spending that looks like Equation 4-5.

$$\mathit{Predicted\,\$\,Spent}\ 2013 = \$50 + 1.1 * (\$\text{Spent } 2012) + \$10 * \text{Age}$$

Equation 4-5: Predicting book spending

Equation 4-5 does not use the Gender variable because it's a categorical variable. (That gets covered in "Incorporating Non-Numeric Attributes into Linear Methods" and is ignored for now.) The predictions generated by Equation 4-5 do not exactly match the Outcomes (actual number) in Table 4-1.

Training Linear Models: Minimizing Errors and More

Finding the values for the β's by hand is not usually the best way, although it's always a good sanity check if you can manage it. In many problems, the size of the problem or the interrelationships between the variables makes guessing the β's impossible. So, the approach taken is to find the multipliers on the attributes (the β's) by solving a minimization problem. The minimization problem is to find the values for the β's that makes the average squared error the smallest (but not zero).

Making the two sides of Equation 4-4 exactly equal usually means the model is overfit. The right side of Equation 4-4 is the predictive model you're going to train. Basically, it says that to make a prediction, you take each attribute, multiply by its corresponding beta, sum these products, and add a constant. *Training the model* means finding the numbers that make up the vector β and the constant, β_0. *Error* is defined as the difference between the actual value of y_i and the prediction o of y_i given by Equation 4-4. The average squared error is used to reduce the individual errors to a single number to be minimized. The square of the error is chosen because it's positive regardless of whether the error is positive or negative and because the square function facilitates some of the math. The formulation of the ordinary least squares regression problem is then to find β_0^*, β^* (the superscript * indicates that these are the best values for β's) that satisfy

$$\beta_0^*, \beta^* = argmin_{\beta_0, \beta}\left(\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \left(x_i{}^*\beta + \beta_0\right)\right)^2\right)$$

Equation 4-6: Minimization problem for OLS

The notation *argmin* means "the arguments that minimize the following expression." The sum is over rows, where a row includes the attribute values and the corresponding labels. The expression inside the $()^2$ is the error between y_i and the linear function that's being used to approximate it. For the predicted $ spent on books in 2013, the expression inside the sum would be the values in the Outcome column minus the prediction calculated from Equation 4-4.

In English, Equation 4-6 says the vector beta star and the constant beta zero star are the values that minimize the expected prediction squared error—that is, the average squared error between y_i and the row of attributes predicted y_i over all data rows (i = 1, . . . , n). The minimization in Equation 4-5 yields the ordinary least squares values for this regression model. This machine learning model is a list of real numbers—the ones included in the vector $\beta^*$and the number β_0^*.

Adding a Coefficient Penalty to the OLS Formulation

The mathematical statement of the penalized linear regression problem is very similar to Equation 4-5. Ridge regression, which you saw in Chapter 3, gives an example of penalized linear regression. Ridge regression adds a penalty term to the basic ordinary least squares problem stated in Equation 4-5. The penalty term for ridge regression is shown in Equation 4-7.

$$\frac{\lambda\beta^T\beta}{2} = \frac{\lambda\left(\beta_1^2 + \beta_2^2 + \ldots + \beta_n^2\right)}{2}$$

Equation 4-7: Penalty applied to coefficients (betas)

The OLS problem in Equation 4-6 was to choose β's to minimize the sum of squared errors. The penalized regression problem adds the coefficient penalty in Equation 4-7 to the right-hand side of Equation 4-6. The minimization is then forced to balance the conflicting goals of minimizing the squared prediction error and the squared values of the coefficients. It is easy to minimize the sum of the squared coefficients by themselves. Just make the coefficient all zero. But that results in large prediction error. Similarly, the OLS solution minimizes the prediction errors by themselves but may result in a large coefficient penalty, depending on how large λ is.

Why does this make sense? To help develop some intuition for why this makes sense, think about the subset selection process that you saw in Chapter 3. Using subset selection eliminated overfitting by discarding some of the attributes, or equivalently by setting their coefficients to zero. Penalized regression does the same thing, but instead of reducing the coefficients of a few attributes all the way to zero like subset selection, penalized regression takes a little coefficient away from all of the attributes. Some limiting cases will also help visualize the approach.

The parameter λ can range anywhere between 0 and plus infinity. If λ=0, the penalty term goes away, and the problem reverts to being an ordinary least squares problem. If $\lambda \to \infty$, the penalty on the β's becomes so severe that it forces them all to zero. (Notice, however, that β_0 is not included in the penalty so the prediction becomes a constant independent of the x's.)

As you saw in the examples in Chapter 3, the ridge penalty can have a similar effect to leaving out some of the attributes. The process is to generate a whole family of solutions to the penalized version of the minimization problem shown in Equation 4-6. That meant solving the penalized minimization problem for a variety of different values of λ. Each of these solutions is then tested on out-of-sample data, and the solution that minimizes the out-of-sample error is used for making real-world predictions. Chapter 3 illustrated this sequence of steps using ridge regression.

Other Useful Coefficient Penalties—Manhattan and ElasticNet

The ridge penalty is not the only useful penalty that can be used for penalized regression. Any metric of vector length will work. You can gauge the length of a vector in a number of ways. Using different measures of length changes important properties of the solution. Ridge regression employed the metric of

Euclidean geometry (that is, the sum of the *squared* β's). Another useful algorithm called Lasso regression employs the metric of taxicab geometry called the *Manhattan length* or *L1 norm* (that is, the sum of the *absolute* β's). Lasso regression has some useful properties.

The difference between ridge regression and Lasso regression is the measure of length that each one uses for penalizing β, the vector of linear coefficients. Ridge uses squared Euclidean distance—the sum of the squares of the components of β. Lasso uses the sum of the absolute values of the components of β—called taxicab or Manhattan distance. The ridge penalty is the squared length of a straight line between zero and the vector space point β (distance as the crow flies). The Lasso penalty is like the distance that a taxicab would have to drive in a city where the streets constrain it to move north-south or east-west only. The lasso penalty is given by the following:

$$\lambda\|\beta_1\| = \lambda\left(|\beta_1| + |\beta_2| + \ldots + |\beta_n|\right)$$

Equation 4-8: Equation for Manhattan distance penalty

The double vertical bars are called norm bars. They are used to denote magnitude for things like vectors and operators. The subscript 1 on the right side of the norm bars denotes l_1 norm, which means the sum of absolute values. You'll also see this written with a capital L_1. Norm bars with a subscript 2 mean square root of the sum of squared values—Euclidean distance. These different coefficient penalty functions cause some important and useful changes in the solutions. One of the main differences is that the Lasso coefficient vector β^* is sparse, meaning that many of the coefficients are zero for large to moderate values of λ. By contrast, the ridge regression β^* is completely populated.

Why Lasso Penalty Leads to Sparse Coefficient Vectors

Figures 4-1 and 4-2 illustrate how this sparsity property stems directly from the form of the coefficient penalty function. These figures are for a problem that has two attributes: x1 and x2.

Both Figure 4-1 and 4-2 have two sets of curves. One set of curves is concentric ellipses that represent the ordinary least squares errors in Equation 4-6. The ellipses represent curves of constant sum squared error. You can think of them as being a topographic map of an elliptical depression in the ground. The error gets smaller for the more central ellipsis, just like the altitude of a depression in the ground gets smaller toward the bottom of the depression. The minimum point for the depression is marked with an x. The point x marks the ordinary least squares solution—where the solution lies if there is no coefficient penalty.

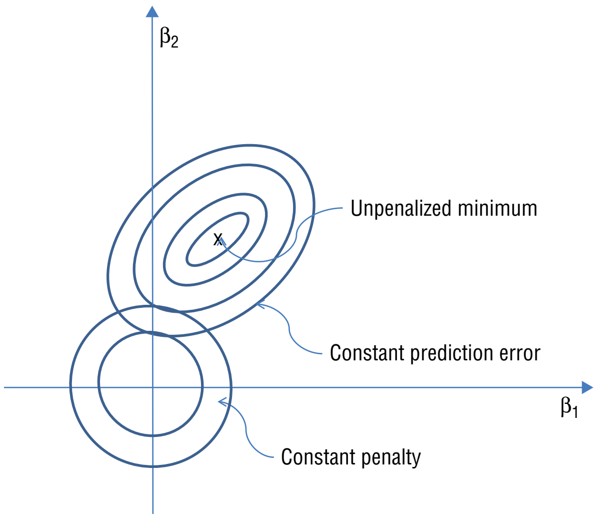

Figure 4-1: Optimum solutions with sum squared coefficient penalty.

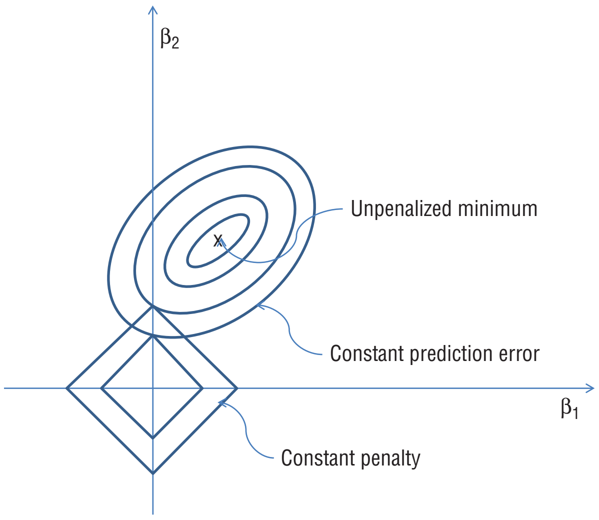

Figure 4-2: Optimum solutions with sum absolute value coefficient penalty.

The other sets of curves in Figures 4-1 and 4-2 represent the coefficient penalties from Equations 4-7 and 4-9—the ridge and Lasso penalties, respectively. In Figure 4-1, the curves representing the coefficient penalty are circles centered at the origin. The set of points where the sum of the squares of β_1 and β_2 is constant defines a circle. The shape of the curves of constant penalty is determined by the nature of the distance measure being used—circles (called hypersphere or l_2 ball in higher dimensions) for sum square penalty function and diamonds (or l_2 ball) for sum of absolute values. Smaller circles (or diamonds) correspond to smaller value for the distance function. The shape is determined by the nature of the penalty function, but the value associated with each curve is determined by the non-negative parameter λ. Suppose that the two curves in Figures 4-1 correspond to sum of squares of β_1 and β_2 equal to 1.0 and 2.0 for the inner and outer circles. Then if $\lambda = 1$, the penalty associated with the two circles is 1 and 2. If $\lambda = 10$, the associated penalties are 10 and 20. The same is true of the diamonds in Figure 4-2. Increasing λ increases the penalty associated with the concentric diamonds in Figure 4-2.

The elliptical rings corresponding to the sum squares of the prediction error also get larger as the rings get farther from the unconstrained minimum, marked by an x in the figure. Minimizing the sum of these two functions, as indicated in Equation 4-6, requires a compromise somewhere in between the minimum for the prediction error and the coefficient penalty. Larger values of λ will pull the compromise closer to the minimum for the penalty (all zero coefficients). Smaller values of λ will pull the minimum closer to the unconstrained minimum prediction error (the x in Figures 4-1 and 4-2).

Here's where the distinction between sum of squared coefficient penalties and sum of absolute value penalties becomes important. The overall minimum for Equations 4-6 or 4-8 will always be at a point where the curve of constant penalty is tangent to the curve of squared prediction error. Figures 4-1 and 4-2 display two examples illustrating this tangency. The important point to make here is that in Figure 4-1 as λ changes and shifts the minimum point, the point of tangency for the sum of squares penalties (the circles) is generally a point that is not on either of the coordinate axes. Neither β_1 nor β_2 is zero. In Figure 4-2, by contrast, the point of tangency for the sum of absolute value stays stuck to the β_2-axis over a range of solutions. Along the β_2-axis, $\beta_1 = 0$.

A sparse coefficient vector is the algorithm's way of telling you that you can completely ignore some of the variables. When λ gets small enough, the best values of β_2 and β_1 will move off the β_2 axis, and both will be nonzero. The fact that a smaller penalty is required to make β_1 non-zero, gives an order to β_2 and β_1. In some sense, β_2 is more important than β_1 because it gets a nonzero coefficient for larger values of λ. Remember that these coefficients multiply attributes. If the coefficient corresponding to an attribute is zero, the algorithm is telling you that attribute is less important than the attributes that are getting nonzero coefficients. By scanning λ from large values to small ones, you can arrange all of the attributes in order of their importance. The next section shows this for a concrete problem and will show Python code that will make explicit the importance comparison between attributes as part of calculating solutions to Equation 4-8.

ElasticNet Penalty Includes Both Lasso and Ridge

Before seeing how to compute these coefficients, you need to know one more generalized statement of the penalized regression problem. This is called the

ElasticNet formulation. The ElasticNet formulation of the penalized regression problem is to use an adjustable blend of the ridge penalty and the Lasso penalty. ElasticNet introduces another parameter, α, that parameterizes the fraction of the total penalty that is the ridge penalty and the fraction that is Lasso penalty. The end point $\alpha = 1$ corresponds to all Lasso penalty and no ridge penalty.

With the ElasticNet formulation, both λ and α must be specified to solve for the coefficients for a linear model. Usually, the approach is to pick a value for α and solve for a range of λ's. You'll see the computational reasons for that later. In many cases, there's not a big performance difference between $\alpha = 1$ and $\alpha = 0$ or some intermediate value of α. Sometimes it will make a big difference, and it behooves you to check to a few different values of α to make sure that you're not sacrificing performance needlessly.

Solving the Penalized Linear Regression Problem

In the preceding section, you saw that determining a penalized linear regression model amounts to solving an optimization problem. A number of general-purpose numeric optimization algorithms will solve the optimization problems in Equations 4-6, 4-8, and 4-11, but the importance of the penalized linear regression problem has motivated researchers to develop specialized algorithms that generate solutions very rapidly. This section covers the basics of these algorithms and runs the code so that you can understand the mechanics of each algorithm. The section goes through the mechanics of two algorithms *least angle regression* or LARS and Glmnet. These two are chosen because they can be related to one another and to some of the methods you have already seen, such as ridge regression and forward stepwise regression. In addition, they are both very fast algorithms to train and are available as part of Python packages. Chapter 5, "Building Predictive Models Using Penalized Linear Methods," will use the Python packages incorporating these algorithms to explore example problems.

Understanding Least Angle Regression and Its Relationship to Forward Stepwise Regression

One very fast, very clever algorithm is the least-angle regression (LARS) algorithm developed by Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (`http://en.wikipedia.org/wiki/Least-angle_regression`).[1] The LARS algorithm can be understood as a refinement to the forward stepwise algorithm that you saw in Chapter 3. The forward stepwise algorithm is summarized here:

Forward Stepwise Regression Algorithm

- Initialize all the β's equal to zero.

At each step

- Find residuals (errors) after using variables already chosen.
- Determine which unused variable best explains residuals and add it to the mix.

The LARS algorithm is very similar. The main difference with LARS is that instead of unreservedly incorporating each new attribute, it only partially incorporates them. The summary for the LARS algorithm is summarized here:

Least Angle Regression Algorithm

- Initialize all β's to zero.

At Each Step

- Determine which attribute has the largest correlation with the residuals.
- Increment that variable's coefficient by a small amount if the correlation is positive or decrement by a small amount if negative.

The LARS algorithm solves a slightly different problem from those listed earlier. However, the solutions it generates are usually the same as Lasso, and when there are differences, the differences are relatively minor. The reason for looking closely at the LARS algorithm is that it is very closely related to Lasso and to forward stepwise regression, and the LARS algorithm is easy to outline and relatively compact to code. By looking at the code for LARS, you'll get an understanding of what goes on inside more general ElasticNet solvers. More important, you'll see the issues and workarounds that accompany penalized regression solvers. Code implementing the LARS algorithm is shown in Listing 4-1.

There are three major sections to the code, described briefly here and then discussed in more detail:

1. Read in the data and headers and form it into a list of lists for the attributes and the labels.
2. Normalize the attributes and the labels.
3. Solve for the coefficients (β_0^*, β^*) that comprise the solution.

Listing 4-1: LARS Algorithm for Predicting Wine Taste—larsWine2.py

```
__author__ = 'mike-bowles'

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

continues

continued

```
#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

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)]

#initialize a vector of coefficients beta
beta = [0.0] * ncols

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


#number of steps to take
nSteps = 350
stepSize = 0.004

for i in range(nSteps):
    #calculate residuals
    residuals = [0.0] * nrows
    for j in range(nrows):
        labelsHat = sum([xNormalized[j][k] * beta[k]
           for k in range(ncols)])
        residuals[j] = labelNormalized[j] - labelsHat

    #calculate correlation between attribute columns from
    #normalized wine and residual
    corr = [0.0] * ncols

    for j in range(ncols):
        corr[j] = sum([xNormalized[k][j] * residuals[k] \
           for k in range(nrows)]) / nrows

    iStar = 0
    corrStar = corr[0]

    for j in range(1, (ncols)):
        if abs(corrStar) < abs(corr[j]):
            iStar = j; corrStar = corr[j]

    beta[iStar] += stepSize * corrStar / abs(corrStar)
    betaMat.append(list(beta))


for i in range(ncols):
    #plot range of beta values for each attribute
    coefCurve = [betaMat[k][i] for k in range(nSteps)]
    xaxis = range(nSteps)
```

continues

continued

```
    plot.plot(xaxis, coefCurve)

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

The first section reads in the entire file, separates off the headers and splits on `";"` to form the headers into a list of attribute names, splits the remaining rows into lists of floats, and segregates the attributes into a list of lists and the labels into a list. Ordinary Python lists are used for these data structures because the algorithm is going to want to iterate through the rows and columns, and Pandas data frames seem slow for this purpose.

The second section uses the same normalization that you saw in Chapter 2, "Understand the Problem by Understanding the Data." In Chapter 2, normalization of the attributes was used to bring attributes into commensurate scales so that they'd plot conveniently and fully occupy the same scale. Normalization is usually done as the first step in penalized linear regression for much the same reason.

Each step in the LARS algorithm increments one of the β's by a fixed amount. If the attributes have different scales, this fixed increment means different things to different attributes. Also, changing the scale on one of the attributes (say from miles to feet) makes the answers come out differently. For these reasons, penalized linear regression packages generally normalize using the common normalization that you saw in Chapter 2. They normalize to zero mean (by subtracting the mean) and unit standard deviation (by dividing the result by standard deviation). Packages will often give you the option of not normalizing, but I've never heard a good reason for not normalizing.

The third and final section solves for β_0^*, β^*. Because the algorithm is running on the normalized variables, there's no need for the intercept β_0^*. That would normally account for any difference between the labels and the weighted attributes. Because all the attributes have been normalized to zero mean, there's no offset between them and no purpose for β_0^*. Notice that two beta-related lists are initialized. One is called *beta* and has the same number of elements as the number of attributes—one weight for each attribute. The other is a matrix-like list of lists that will house a list of betas for each step in the LARS algorithm. This gets into a key concept with penalized linear regression and modern machine learning algorithms in general.

How LARS Generates Hundreds of Models of Varying Complexity

Modern machine learning algorithms in general, and penalized linear regression in particular, generate families of solutions, not just single solutions. Look back at Equations 4-6, 4-8, and 4-11. On the left side of those equations are the ,β-'.*s*, and on the right hand side are all numeric values that are fixed by the data available for the problem with one exception. In Equations 4-6 and 4-8, there is a parameter λ that has to be determined some other way. As was pointed out in the discussion of those equations, when $\lambda=0$, the problems reduce to ordinary least squares regression, and when $\lambda \to \infty$, $\beta^* \to 0$. So, the β's depend on the parameter λ in the problems stated in Equations 4-6, 4-8, and 4-11.

The LARS algorithm doesn't explicitly deal with λ values, but it has the same effect. The LARS algorithm starts with β's equal to zero and then adds a small increment to whichever of the β's will reduce the error the most. The small increment that's added increases the sum of absolute values of the β's by the amount of the increment. If the increment is small and if it's spent on the best of the attributes, the process has the effect of solving the minimization problem in Equation 4-8. You can trace the evolution of this process in Listing 4-1.

The basic iteration is just a few lines of code at the beginning of the for-loop iterating for `nSteps`. The starting point for the iteration is a value for the β's. On the first pass, those are all set to zero. On subsequent passes, they come from the result of the last pass. There are two steps in the iteration. First, the β's are used to calculate residuals. The term *residuals* means the difference between observed outcome and predicted outcome. In this case the predictive method consists of multiplying each attribute times a corresponding element from β and then summing the products. The second step is to find the correlation between each of the attributes and the residuals to determine which attribute will contribute the most to reducing the residual (error). The correlation between two variables is the product of their variations from their means normalized by their individual standard deviations.

Variables that are scaled versions of one another will have correlations of plus one or minus one depending on whether the scaling between them is positive or negative. If two variables vary independently of one another, their correlation is zero. The Wikipedia page on correlation, `http://en.wikipedia.org/wiki/Correlation_and_dependence`, gives good illustrations of variables having other degrees of correlation with one another. The list named corr contains the result of the calculation for each attribute. You may notice that strictly speaking the code omits calculation of the standard deviation of the mean, residuals, and normalized attributes. That works here because the attributes have been normalized to all have standard deviation one and because the resulting values are going to be used to find the biggest correlation and multiplying all the values by a constant won't change that order.

Once the correlations are calculated, it's a simple matter to determine which attribute has the largest correlation with the residuals (largest in absolute value). The corresponding element from the list of β's is incremented by a small amount. The increment is positive if the correlation is positive and negative otherwise. The new value of the β's is then used to rerun the iteration.

The net result from the LARS algorithm are the coefficient curves shown in Figure 4-3. The way to view these is to imagine a point along the "steps taken" axis in the graph. At that point, a vertical line will pass through all the coefficient curves. The values at which the vertical line intersects the coefficient curves are the coefficients at that step in the evolution of the LARS algorithm. If 350 steps are used to generate the curves, there are 350 sets of coefficients. Each one optimizes Equation 4-8 for some value of λ. That raises the question of which one should you use. That question will be addressed shortly.

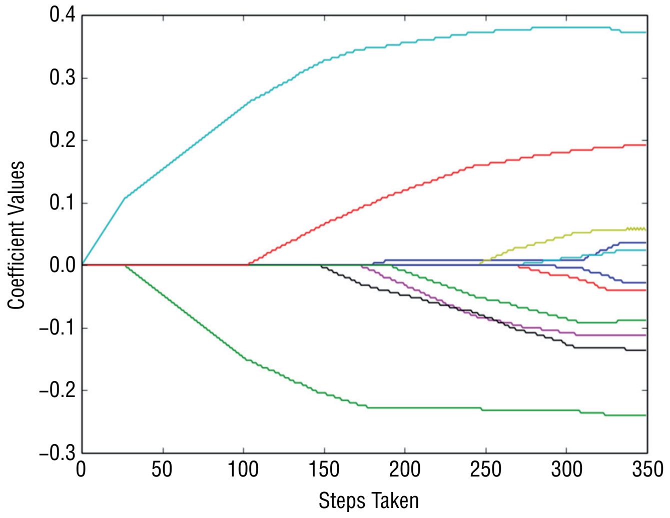

Figure 4-3: Coefficient curves for LARS regression on wine data.

Notice that for the first 25 steps or so, only one of the coefficients is nonzero. This is the sparsity property that comes with Lasso regression. The coefficient that is the first to move off zero is alcohol; for a while, that's the only variable being used by LARS regression. Then a second variable comes into play. This process continues until all the variables are being used in the solution. The order in which coefficients move off zero can be used as an indication of the rank order of importance of the variables. If you had to discard a variable, you'd want to discard one that came in last rather than the one that came in first.

THE IMPORTANCE OF IMPORTANCE

This property of indicating the importance rank of the variables is an important feature of penalized regression methods. It makes them a handy tool to use early in your development process because they'll help you make decisions about what variables to keep and which ones to discard—a process called feature engineering. You'll see later that tree ensembles also yield measures of variable importance. Not all machine learning methods give this sort of information. You could always generate the ordering by trying all combinations of one variable, then two variables, and so on. But even with the mere 10 attributes in the wine data, it's prohibitive to make the 10 factorial training passes required to try all possible subsets.

Choosing the Best Model from the Hundreds LARS Generates

Now you've got 350 possible solutions to the problem of predicting wine taste score from the chemical properties of the wine. How do you choose the best one? To choose which of the curves you'll use, you need to determine how each of the 350 choices performs. As discussed in Chapter 3, *performance* means performance on out of sample data. Chapter 3 outlined several methods for holding out data from the training process to use it to determine performance. Listing 4-2 shows the code for performing 10-fold cross-validation to determine the best set of coefficients to deploy.

Ten-fold cross-validation is the process of dividing the input data into 10 more or less equal groups, removing one of the groups from the data, training on the remainder, and then testing on the removed group. By cycling through all 10 of the groups and removing them one at a time for testing, you can develop a good estimate of the error and of the estimate's variability.

Listing 4-2: 10-Fold Cross-Validation to Determine Best Set of Coefficients—larsWineCV.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import datasets, linear_model
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)
```

continues

continued

```
#Normalize columns in x and labels

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 calculated 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)]

#Build cross-validation loop to determine best coefficient values.

#number of cross-validation folds
nxval = 10

#number of steps and step size
nSteps = 350
stepSize = 0.004

#initialize list for storing errors.
errors = []
for i in range(nSteps):
    b = []
    errors.append(b)


for ixval in range(nxval):
    #Define test and training index sets
```

```
idxTest = [a for a in range(nrows) if a%nxval == ixval*nxval]
idxTrain = [a for a in range(nrows) if a%nxval != ixval*nxval]

#Define test and training attribute and label sets
xTrain = [xNormalized[r] for r in idxTrain]
xTest = [xNormalized[r] for r in idxTest]
labelTrain = [labelNormalized[r] for r in idxTrain]
labelTest = [labelNormalized[r] for r in idxTest]

#Train LARS regression on Training Data
nrowsTrain = len(idxTrain)
nrowsTest = len(idxTest)

#initialize a vector of coefficients beta
beta = [0.0] * ncols

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

for iStep in range(nSteps):
    #calculate residuals
    residuals = [0.0] * nrows
    for j in range(nrowsTrain):
        labelsHat = sum([xTrain[j][k] * beta[k]
           for k in range(ncols)])
        residuals[j] = labelTrain[j] - labelsHat

    #calculate correlation between attribute columns
    #from normalized wine and residual
    corr = [0.0] * ncols

    for j in range(ncols):
        corr[j] = sum([xTrain[k][j] * residuals[k] \
           for k in range(nrowsTrain)]) / nrowsTrain

    iStar = 0
    corrStar = corr[0]

    for j in range(1, (ncols)):
        if abs(corrStar) < abs(corr[j]):
            iStar = j; corrStar = corr[j]

    beta[iStar] += stepSize * corrStar / abs(corrStar)
    betaMat.append(list(beta))

    #Use beta just calculated to predict and accumulate out of
    #sample error - not being used in the calc of beta
    for j in range(nrowsTest):
        labelsHat = sum([xTest[j][k] * beta[k] for k in range
```

continues

continued

```
            (ncols)])
            err = labelTest[j] - labelsHat
            errors[iStep].append(err)

cvCurve = []
for errVect in errors:
    mse = sum([x*x for x in errVect])/len(errVect)
    cvCurve.append(mse)

minMse = min(cvCurve)
minPt = [i for i in range(len(cvCurve)) if cvCurve[i] == minMse ][0]
print("Minimum Mean Square Error", minMse)
print("Index of Minimum Mean Square Error", minPt)

xaxis = range(len(cvCurve))
plot.plot(xaxis, cvCurve)

plot.xlabel("Steps Taken")
plot.ylabel(("Mean Square Error"))
plot.show()

Printed Output:
('Minimum Mean Square Error', 0.5873018933136459)
('Index of Minimum Mean Square Error', 311)
```

Mechanizing Cross-Validation for Model Selection in Python Code

The code in Listing 4-2 begins similarly to the code in Listing 4-1. The differences become clear at the cross-validation loop that is looping `nxval` times. In this case `nxval = 10`, but it could be set to other values as well. The tradeoffs with how many folds to use are that smaller numbers of folds mean that you're training on less of the data. If you take 5 folds, then you're leaving out 20% each training pass. If you take 10 folds, you're only leaving out 10%. As you saw in Chapter 3, training on less data causes deterioration in the accuracy your algorithm will achieve. However, taking more folds means making more passes through the training process. That can be cumbersome in terms of the clock or calendar time required for training.

Just ahead of the cross-validation loop, an error list gets initialized. This error list will consist of a list of errors for each step in the evolution of the LARS algorithm. It will accumulate the errors for each step over all 10 of the cross-validation folds. Just inside the cross-validation loop, you'll see definition of training and test sets. I typically use a modulus function to define these sets unless there's some reason not to. For example, sometimes you may need to do what's called *stratified sampling*. Suppose that you're trying to build a classifier on data that are unbalanced, so there are very few of one of the classes. You want for the training sets to be representative of the full data set. You may need to segregate the data by classes so that the classes are represented in both in-sample and out-of-sample data.

You may prefer to use a random function to define training and test sets. You do need to be aware of any patterning in the data set that would interact with the sampling process adversely (that is, if observations are not exchangeable). For example, if data were taken daily during the work week, then using the modulus function with five-fold cross-validation might result in one set having all the Mondays and another having all the Tuesdays, and so on.

Accumulating Errors on Each Cross-Validation Fold and Evaluating Results

Once the training and test sets are defined along with a few constants, the iteration of the LARS algorithm begins. This is very similar to the process defined in Listing 4-1, with a couple of important differences. First, the basic iteration of the algorithm is carried out on the training set instead of the full data set and second, at each step in the iteration and for each cross-validation fold the current values of the β's are used along with the test attributes and test labels to ascertain the error on the test set for that step. You'll see that calculation at the bottom of the cross-validation loop. Each time β is updated, it is applied to the test data, and the error is accumulated in the appropriate list in "error." It's a simple matter to then square and average each of the lists in error. This produces a curve of the mean square error (MSE) at each iteration, averaged over all 10 of the cross-validation folds.

You might worry whether the test data is being used properly. It's always important to be vigilant about letting the test data leak into the training process. There are numerous ways to trick oneself into violating this necessity. In this case, you'll notice that the test data is not used in the calculation of the increments of β. Only the training data is being used there.

Practical Considerations with Model Selection and Training Sequence

The curve of MSE versus number of steps in the LARS iteration is shown in Figure 4-4. This curve exhibits a fairly common pattern. It decreases more or less monotonically over its whole range. Strictly speaking, it does have a minimum point at around 311, as indicated in the associated printed output from the program. But the graph shows that the minimum is fairly weak, not very sharp. In some cases, this curve will have a sharp minimum at some point and will increase markedly to the right and left of the minimum. You use the result of cross-validation to determine which of the 350 solutions generated by LARS should be used for making predictions. In this case, the minimum is at step 311. The 311th set of β's would be the coefficients to deploy. When there's any ambiguity about the best solution to deploy, it's usually best to deploy the more conservative solution. More conservative for penalized regression means the one with smaller coefficient values. By convention, out-of-sample performance is usually portrayed with the less-complex models on the left and the more-complex models on the right. Less-complex models have better generalization error; that is, they perform more predictably on new data. The more conservative model would be the one more to the left side of the out of sample performance graph.

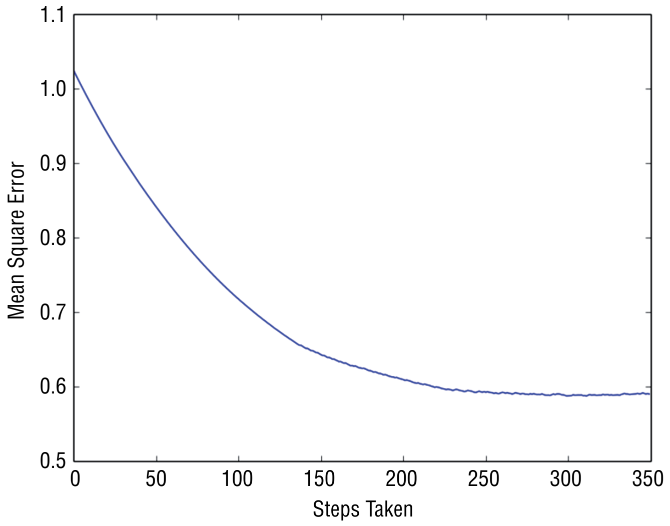

Figure 4-4: Cross-validated mean square error for LARS on wine data.

This description of the LARS algorithm and of the cross-validation process has gone through training the algorithm on the whole data set first, then running cross-validation second. In practice, you'll probably first run cross-validation and then train the algorithm on the whole data set. The purpose of cross-validation is to determine what level of MSE (or other) performance you'll be able to achieve and to learn how complicated a model your data set will sustain. If you recall, Chapter 3 discussed the issues of data set size and model complexity. Cross-validation (or other process for setting aside data to get a sound estimate of performance) is how you determine the best model complexity for the model you will deploy. You determine the complexity but not the specific model (that is, not the specific set of β's). As you can see in Listing 4-2, with 10-fold cross-validation, you've actually trained 10 models, and there's no way to decide among the 10. Best practice is to train on the full data set and to use the cross-validation results to determine which of the models determine which of the models to deploy. In the example shown in Code Listing 4-2, cross-validation gives a minimum MSE of 0.59 at the 311th step in the training process. The coefficient curves in Figure 4-5 were trained on the full data set. The digression into cross-validation was motivated by not knowing which of the 350 sets of coefficients represented in Figure 4-5 should be deployed. Cross-validation has yielded a sound estimate of the MSE and tells us to deploy the 311th model from training on the full data set.

Using Glmnet: Very Fast and Very General

The glmnet algorithm was developed by Professor Jerome Friedman and his colleagues at Stanford.[2] The glmnet algorithm solves the ElasticNet problem given by Equation 4-11. Recall that the ElasticNet problem incorporates a generalization of the penalty function that includes both the Lasso penalty (sum of absolute values) and the ridge penalty (sum of squares). ElasticNet has a parameter λ that determines how heavily the coefficient penalty is penalized compared to the fit error. It also has a parameter that determines how close the penalty is to ridge ($\alpha=0$) or Lasso ($\alpha=1$). The glmnet algorithm yields the full coefficient curves, similar to the LARS algorithm. Whereas the LARS algorithm accumulates quanta of coefficient into the β's to drive the curves forward, the glmnet algorithm makes steady reductions in the λ's to drive the coefficient curves forward. Equation 4-9 shows the key equation from Friedman's paper—the key iterative equation for the coefficients that solve Equation 11—the ElasticNet equation.

$$\tilde{\beta}_j \leftarrow \frac{S\left(\frac{1}{m}\sum_{i=1}^{m} x_{ij} r_i + \tilde{\beta}_j ,\lambda\alpha\right)}{1+\lambda(1-\alpha)}$$

Equation 4-9: Coordinate-wise update for glmnet

Equation 4-9 is a combination of Equations 5 and 8 in Friedman's paper (for those of you who would like to follow the math). It looks complicated, but a little inspection will reveal some similarities and relationships to the LARS method that you saw in the last section.

Comparison of the Mechanics of Glmnet and LARS Algorithms

Equation 4-9 gives the basic update equation for the β's. The update equation for LARS was "find the attribute with the largest magnitude correlation with the residual and increment (or decrement) its coefficient by a small fixed amount." The updated Equation 4-9 is a little more involved. It has an arrow instead of an equals sign. The arrow means something like "gets mapped to." Notice that $\tilde{\beta}_j$ appears on both sides of the arrow. On the right side of the arrow is the old value of $\tilde{\beta}_j$, and on the left side (the direction the arrow points) is the new value of $\tilde{\beta}_j$. After several passes through, the iteration inferred in 4-12, $\tilde{\beta}_j$ stops changing. (More precisely, the change becomes insignificant.) Once $\tilde{\beta}_j$ stops changing, the algorithm has arrived at a solution for the given values of λ and α. It's time to move to the next point in the coefficient curve.

The first thing to notice is the expression $x_{ij}r_i$ inside the sum. The sum of $x_{ij}r_i$ over i (that is over rows of data) yields the correlation between the jth attribute and the residual. Recall that with LARS regression at each step through the algorithm each attribute was correlated against the residuals. In the LARS algorithm, those correlations were tested to see which attribute had the biggest correlation with the residual, and the coefficient corresponding to the attribute with the highest correlation was incremented. With the glmnet algorithm, the correlation is used somewhat differently.

With glmnet, the correlation between the residuals is used to calculate how much each coefficient ought to be changed in magnitude. But the result passes through the function S() before resulting in a change in $\tilde{\beta_j}$. The function S() is the Lasso coefficient shrinkage function. It is plotted in Figure 4-5. As you can see in Figure 4-5, if the first input is smaller than the second, the output is zero. If the first input is larger than the second, the output is the first input reduced in magnitude by the second. This is called a soft limiter.

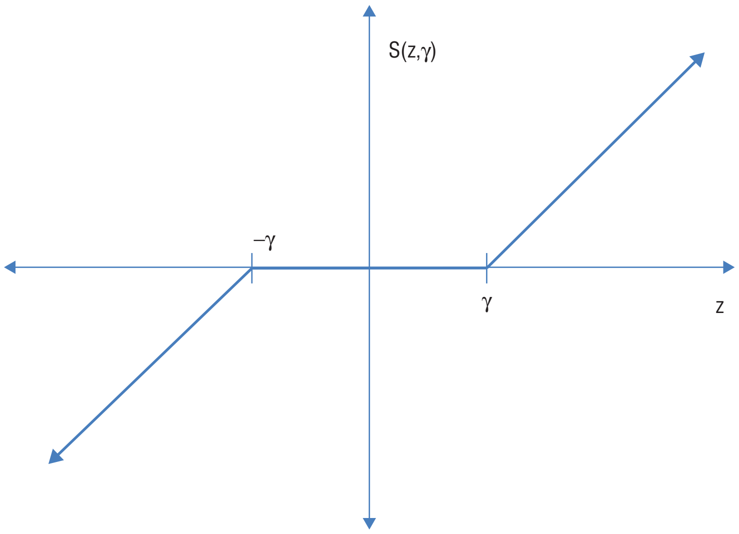

Figure 4-5: Plot of S() function

Listing 4-3 shows code for the glmnet algorithm. You can see in the code how Equation 4-12, for updating the β's, is used to generate ElasticNet coefficient curves. The code in Listing 4-3 is annotated with equation number from Friedman's paper. The paper is very accessible, and you can refer to it to get more mathematical details if you're interested.

Initializing and Iterating the Glmnet Algorithm

The iteration starts with a large value of λ. It begins with a value for λ that is large enough to make all the β's zero. You can see how to calculate the starting value for λ by reference to Equation 4-9. The function S() in Equation 4-12 gives zero for output if its first input (the correlation of $x_{ij}r_i$) is less than the second—λα. The iteration starts with all the β's equal to zero, so the residual is equal to the raw labels. The code for determining the starting lambda calculates the correlations for each of the attributes and the labels, finds the largest in magnitude, and then solves for the value of λ that makes the largest correlation just equal λα. That is the largest value of λ that results in all zero β's.

Then the iteration begins by reducing λ. This is accomplished by multiplying λ by a number slightly less than one. Friedman suggests that the multiplier be selected so that $\lambda^{100} = 0.001$. That gives a value of roughly 0.93. If the algorithm runs for a long time without converging, then the multiplier on λ needs to be made closer to 1. In Friedman's code, the mechanism for accomplishing this is to increase the number of steps from 100 to, say, 200 so that it takes 200 steps to reduce the starting λ to 0.001 of its starting value. In the Listing 4-3, you've got control of the multiplier directly. The coefficient curves are shown in Figure 4-8.

Listing 4-3: Glmnet Algorithm—glmnetWine.py

```
__author__ = 'mike-bowles'

import urllib2
import numpy
from sklearn import datasets, linear_model
from math import sqrt
import matplotlib.pyplot as plot
def S(z, gamma):
    if gamma >= abs(z):
        return 0.0
    return (z/abs(z))*(abs(z) - gamma)

#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

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

#calculate means and variances
xMeans = []
```

continues

continued

```
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)]

#select value for alpha parameter

alpha = 1.0

#make a pass through the data to determine value of lambda that
# just suppresses all coefficients.
#start with betas all equal to zero.


xy = [0.0]*ncols
for i in range(nrows):
    for j in range(ncols):
        xy[j] += xNormalized[i][j] * labelNormalized[i]

maxXY = 0.0
for i in range(ncols):
    val = abs(xy[i])/nrows
    if val > maxXY:
        maxXY = val

#calculate starting value for lambda
lam = maxXY/alpha

#this value of lambda corresponds to beta = list of 0's
#initialize a vector of coefficients beta
beta = [0.0] * ncols

#initialize matrix of betas at each step
```

```
betaMat = []
betaMat.append(list(beta))

#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):
    #make lambda smaller so that some coefficient becomes non-zero
    lam = lam * lamMult

    deltaBeta = 100.0
    eps = 0.01
    iterStep = 0
    betaInner = list(beta)
    while deltaBeta > eps:
        iterStep += 1
        if iterStep > 100: break

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

            xyj = 0.0
            for i in range(nrows):
                #calculate residual with current value of beta
                labelHat = sum([xNormalized[i][k]*betaInner[k]
                                for k in range(ncols)])
                residual = labelNormalized[i] - labelHat

                xyj += xNormalized[i][iCol] * residual

            uncBeta = xyj/nrows + betaInner[iCol]
            betaInner[iCol] = S(uncBeta, lam * alpha) / (1 +
                                            lam * (1 - alpha))

        sumDiff = sum([abs(betaInner[n] - betaStart[n])
                       for n in range(ncols)])
        sumBeta = sum([abs(betaInner[n]) for n in range(ncols)])
        deltaBeta = sumDiff/sumBeta
    print(iStep, iterStep)
    beta = betaInner

    #add newly determined beta to list
    betaMat.append(beta)

    #keep track of the order in which the betas become non-zero
    nzBeta = [index for index in range(ncols) if beta[index] != 0.0]
    for q in nzBeta:
```

continues

continued

```
        if (q in nzList) == False:
            nzList.append(q)

#print out the ordered list of betas
nameList = [names[nzList[i]] for i in range(len(nzList))]
print(nameList)

nPts = len(betaMat)
for i in range(ncols):
    #plot range of beta values for each attribute
    coefCurve = [betaMat[k][i] for k in range(nPts)]
    xaxis = range(nPts)
    plot.plot(xaxis, coefCurve)

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

#Printed Output:
#['"alcohol"', '"volatile acidity"', '"sulphates"',
#'"total sulfur dioxide"', '"chlorides"', '"fixed acidity"', '"pH"',
#'"free sulfur dioxide"', '"residual sugar"', '"citric acid"',
#'"density"']
```

Figure 4-8 shows the coefficient curves generated by Listing 4-3. The curves look similar in character to those generated by LARS and shown in Figure 4-6—similar but not identical. LARS and Lasso often give the same curves, but sometimes give somewhat different results. The only way to tell which one is superior is to try them both against out-of-sample data and see which one gives the best performance.

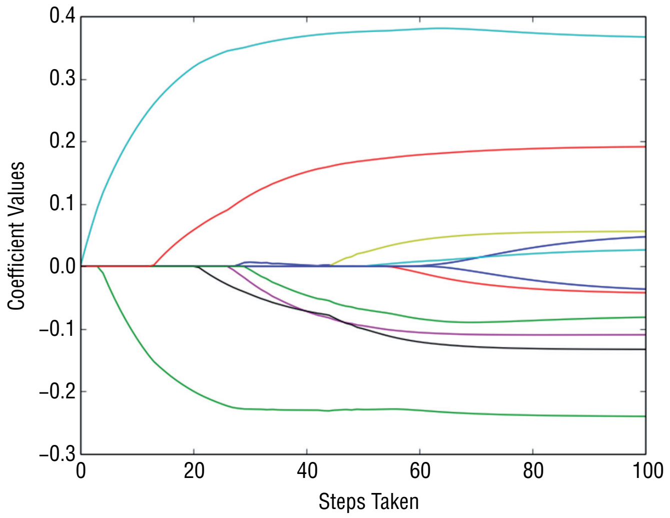

Figure 4-6: Coefficient curves for glmnet models for predicting wine taste

The development process for a Lasso model is the same as for LARS. Use one of the methods described in Chapter 3 for testing on out-of-sample data (n-fold cross-validation, for example). Use the results on out-of-sample data to determine the optimum model complexity. Then train on the full data set to build coefficient curves and pick the step in the coefficient curves that out-of-sample testing shows to be the optimum.

This section has gone through two solution approaches for solving the minimization problems that define penalized linear regression models. You've seen how these two methods work algorithmically, how they relate to one another and what the code looks like to implement them. This should give you a firm foundation for using the packages available in Python that implement these algorithms. It also puts you in a good position to understand various extensions to the models that will be covered in the next section and that will be used in the examples that you'll see in Chapter 5.

Extensions to Linear Regression with Numeric Input

So far, the development has focused on regression problems—problems where the outcomes being predicted take real number values. How can the machinery discussed be applied to classification problems—problems where the outcomes take two (or more) discrete values like "click" or "not click"? There are several ways to extend what you've seen so far to cover classification problems.

Solving Classification Problems with Penalized Regression

For binary classification problems, you'll often get good results by coding the binary values as real numbers. This simple procedure codes one of the two binary values as a 1 and the other as a 0 (or +1 and –1). With that simple arrangement, the list of labels becomes a list of real numbers, and the algorithms already discussed can be employed. This is often a good alternative even though there are more sophisticated approaches. This simple coding approach usually trains faster than more sophisticated approaches and that can be important.

Listing 4-4 gives an example of using the method of substituting numeric 0 or 1 labels for class membership in the rocks versus mines data set. You'll recall from Chapter 2 that the rocks versus mines data set presents a classification problem. The data set comes from an experiment to determine if sonar can be used to detect unexploded mines left in the water. Various other objects besides mines will reflect the sonar's sound waves. The prediction problem is to determine whether the reflected waves come from an unexploded mine or from rocks on the sea floor.

The sonar in the experiment uses what's called a chirped waveform. A chirped waveform is one that rises (or falls) in frequency over the duration of the transmitted sonar pulse. The 60 attributes in the rocks versus mines data set are the returned pulse sampled at 60 different times, which correspond to 60 different frequencies in the chirped pulse.

Listing 4-4 demonstrates how to convert the classification labels R and M into 0.0 and 1.0 to convert the problem into an ordinary regression problem. The code then uses the LARS algorithm to build a classifier. Listing 4-4 goes through a single pass on the full data set. As discussed in the last section, you'll want to use cross-validation or some other holdout procedure to choose the optimal model complexity. Chapter 5 goes through those design steps and performance comparisons on this data set. The point here is for you to see how to apply the regression tools you've already seen to a classification problem.

Listing 4-4: Converting a Classification Problem to an Ordinary Regression Problem by Assigning Numeric Values to Binary Labels

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

#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])



#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
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)]

#initialize a vector of coefficients beta
beta = [0.0] * ncol

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


#number of steps to take
nSteps = 350
stepSize = 0.004
nzList = []

for i in range(nSteps):
    #calculate residuals
    residuals = [0.0] * nrow
    for j in range(nrow):
        labelsHat = sum([xNormalized[j][k] * beta[k]
           for k in range(ncol)])
        residuals[j] = labelNormalized[j] - labelsHat
```

continues

continued

```
    #calculate correlation between attribute columns from
    #normalized X and residual
    corr = [0.0] * ncol

    for j in range(ncol):
        corr[j] = sum([xNormalized[k][j] * residuals[k]
           for k in range(nrow)]) / nrow

    iStar = 0
    corrStar = corr[0]

    for j in range(1, (ncol)):
        if abs(corrStar) < abs(corr[j]):
            iStar = j; corrStar = corr[j]

    beta[iStar] += stepSize * corrStar / abs(corrStar)
    betaMat.append(list(beta))


    nzBeta = [index for index in range(ncol) if beta[index] != 0.0]
    for q in nzBeta:
        if (q in nzList) == False:
            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(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:
#['V10', 'V48', 'V44', 'V11', 'V35', 'V51', 'V20', 'V3', 'V21', 'V15',
# 'V43', 'V0', 'V22', 'V45', 'V53', 'V27', 'V30', 'V50', 'V58', 'V46',
# 'V56', 'V28', 'V39']
```

Figure 4-7 shows the coefficient curves developed by the LARS algorithm. They are similar in character to the curves you saw for the wine taste prediction problem. However, there are more curves because the rocks versus mines data set has more attributes. (The rock versus mines data has 60 attributes and 208 rows of data.) From the discussion in Chapter 3, you might expect that the optimum solution won't use all the attributes. You'll see how that tradeoff turns out in Chapter 5, which concentrates on solutions to this and other problems and comparisons between different approaches.

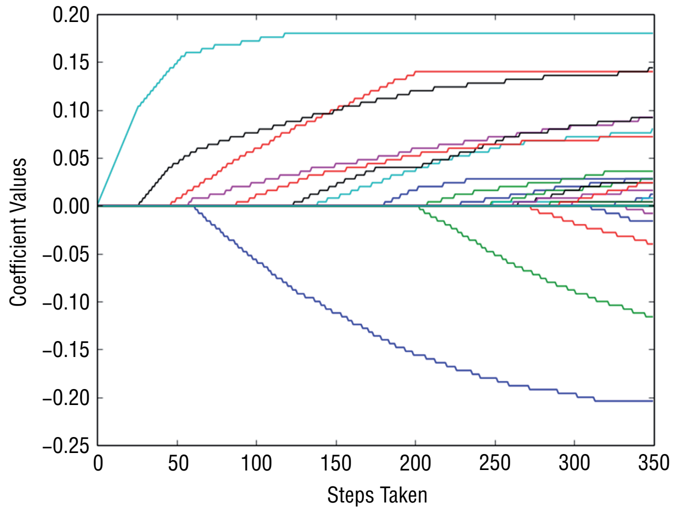

Figure 4-7: Coefficient curves for rocks versus mines classification problem solved by converting to labels

Another approach is to formulate the problem in terms of the likelihoods of the two outcomes in the problem. That leads to what's called *logistic regression*. The glmnet algorithm can be cast in that framework, and Friedman's original paper goes through the development of the logistic regression version of glmnet and of its extension to multiclass problems—problems with more than two discrete outcomes. You'll see the use of the binary and multiclass versions of the algorithm in Chapter 5.

Working with Classification Problems Having More Than Two Outcomes

Some problems require deciding among several alternatives. For example, say you show a visitor to your website several links. The visitor may click on any one of the several links, click the back button, or exit the site entirely. There are several alternatives that aren't ordered like the integer wine taste scores are. A taste score of 4 naturally fits between 3 and 5, and if changing an attribute (like alcohol) makes the score go from 3 to 4, changing it some more seems likely to move the score further in the same direction. Alternative actions a site visitor will take have no such order. This is called a multiclass classification problem.

You can always handle a multiclass problem with an algorithm for binary classification. The technique is called *one versus all* or *one versus the rest*, and the names give you some idea of how the approach works. Basically you pose your multiclass problem as several binary problems. For the example, you could predict whether the visitor would leave the site or choose another option. Another binary classification problem is to predict whether the user would click the back button or take any of the rest of the options available. You'll wind up with as many binary classification problems as you have alternative outcomes. The binary classifiers all give numeric values, like the LARS classifier in Listing 4-4. The outcome that has the largest one-versus-all value is the winner. Chapter 5 implements this method for the glass data set, where there are six different possible outcomes.

Understanding Basis Expansion: Using Linear Methods on Nonlinear Problems

By their nature, linear methods assume classification and regression predictions can be expressed as a linear combination of the attributes that are available to the designer. What if you have reason to suspect that a linear model isn't enough? You can get a linear model to work with strong nonlinearities by using what's called basis expansion. The basic idea behind basis expansion is that the nonlinearities in your problem can be approximated as polynomials of the attributes (or sum of other nonlinear functions of the attributes); then you can add attributes that are powers of the original attributes and let a linear method determine the best set of coefficients for the polynomial.

To get a concrete idea of how this would work, look at the code in Listing 4-5. Listing 4-5 starts with the wine taste data set. If you recall, the linear models that were produced earlier in this chapter both found that alcohol was the most important attribute in determining wine taste. It occurs to you that the relationship might not be a straight line, but might roll off for really high alcohol content and for really low alcohol content.

Listing 4-5 shows you how to test this notion.

Listing 4-5: Basis Expansion for Wine Taste Prediction

```
__author__ = 'mike-bowles'

import urllib2
import matplotlib.pyplot as plot
from math import sqrt, cos, log

#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)


#extend the alcohol variable (the last column in that attribute matrix
xExtended = []
alchCol = len(xList[1])


for row in xList:
    newRow = list(row)
    alch = row[alchCol - 1]
    newRow.append((alch - 7) * (alch - 7)/10)
    newRow.append(5 * log(alch - 7))
    newRow.append(cos(alch))
    xExtended.append(newRow)

nrow = len(xList)
v1 = [xExtended[j][alchCol - 1] for j in range(nrow)]

for i in range(4):
    v2 = [xExtended[j][alchCol - 1 + i] for j in range(nrow)]
    plot.scatter(v1,v2)

plot.xlabel("Alcohol")
plot.ylabel(("Extension Functions of Alcohol"))
plot.show()
```

The code reads in the data as before. Right after reading in the data (and before it is normalized), the code runs through the rows of data that it's read, adds a few new elements to the row, and then appends the new expanded row to a new set of attributes. The new elements that are appended are all functions of the alcohol attribute in the original data. For example, the first new attribute is ((alch - 7) * (alch - 7)/10), where alch is the alcohol level in the row.

The constants 7 and 10 were introduced so that the resulting new attributes would all plot nicely on one plot. Basically, the new attribute is alcohol squared.

The next step in the process is to take the expanded set of attributes and build a linear model using the tools already developed in this chapter (or another of the methods available for building linear models). Whatever algorithm is used for building a linear model, the model will consist of multipliers (or coefficients) for each of the attributes, including the new ones. If the functions used in the expansion are all powers of the original variable, the linear model yields coefficients in a polynomial function of the original variable. By choosing different functions for the expansion, other function series can be constructed.

Figure 4-8 illustrates the functional dependence of the new attributes (and the original attribute) on the original attribute. You can see the squared, logarithmic, and sinusoidal behavior of the selection of functions in the expansion.

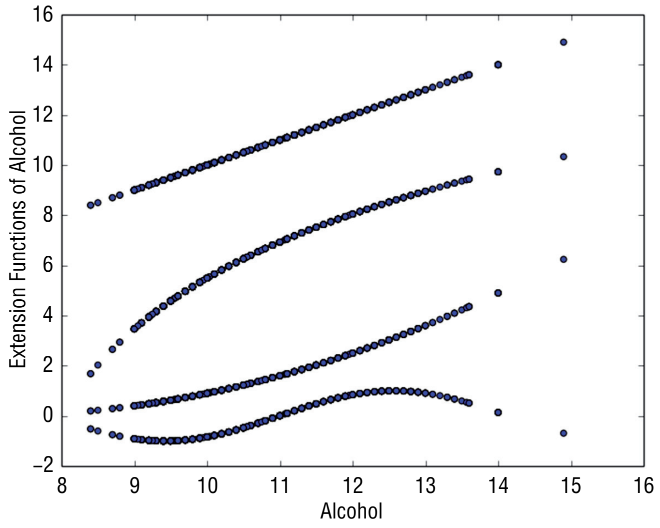

Figure 4-8: Functions generated to expand wine attribute session

Incorporating Non-Numeric Attributes into Linear Methods

Penalized linear regression (and other linear methods) require numeric attributes. What if your problem has some non-numeric attributes (also called categorical or factor attributes)? A familiar example would be a gender attribute where the possibilities are male and female. The standard method for converting categorical variables to numeric is to code them into several new columns of attribute data. If an attribute has N possible values, it gets coded into N - 1 new columns of data as follows. Identify N - 1 columns of data with N - 1 of the N attributes. In each row enter a 1 in the ith column if the row takes the ith possible value of the categorical variable. Put zeros in the other columns. If the row takes the Nth value of the categorical variable, all the entries will be zero.

Listing 4-6 shows how this technique can be applied to the abalone data set. The task with the abalone data set is to predict the age of abalone from various physical measurements.

Listing 4-6: Coding Categorical Variable for Penalized Linear Regression - Abalone Data—larsAbalone.py

```
__author__ = 'mike_bowles'

import urllib2
from pylab import *
import matplotlib.pyplot as plot

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)

names = ['Sex', 'Length', 'Diameter', 'Height', 'Whole weight', \
    'Shucked weight', 'Viscera weight', 'Shell weight', 'Rings']

#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)
```

continues

continued

```
namesCoded = ['Sex1', 'Sex2', 'Length', 'Diameter', 'Height', \
   'Whole weight', 'Shucked weight', 'Viscera weight', \
   'Shell weight', 'Rings']

nrows = len(xCoded)
ncols = len(xCoded[1])

xMeans = []
xSD = []
for i in range(ncols):
    col = [xCoded[j][i] for j in range(nrows)]
    mean = sum(col)/nrows
    xMeans.append(mean)
    colDiff = [(xCoded[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 xCoded
xNormalized = []
for i in range(nrows):
    rowNormalized = [(xCoded[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)]

#initialize a vector of coefficients beta
beta = [0.0] * ncols

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


#number of steps to take
nSteps = 350
stepSize = 0.004
nzList = []

for i in range(nSteps):
    #calculate residuals
    residuals = [0.0] * nrows
    for j in range(nrows):
        labelsHat = sum([xNormalized[j][k] * beta[k]
           for k in range(ncols)])
        residuals[j] = labelNormalized[j] - labelsHat

    #calculate correlation between attribute columns from
    #normalized wine and residual
    corr = [0.0] * ncols

    for j in range(ncols):
        corr[j] = sum([xNormalized[k][j] * residuals[k]
           for k in range(nrows)]) / nrows

    iStar = 0
    corrStar = corr[0]

    for j in range(1, (ncols)):
        if abs(corrStar) < abs(corr[j]):
            iStar = j; corrStar = corr[j]

    beta[iStar] += stepSize * corrStar / abs(corrStar)
    betaMat.append(list(beta))


    nzBeta = [index for index in range(ncols) if beta[index] != 0.0]
    for q in nzBeta:
        if (q in nzList) == False:
            nzList.append(q)

nameList = [namesCoded[nzList[i]] for i in range(len(nzList))]

print(nameList)
for i in range(ncols):
    #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- larsAbaloneOutput.txt]
['Shell weight', 'Height', 'Sex2', 'Shucked weight', 'Diameter', 'Sex1']
```

The first attribute is the gender of the abalone, which takes three values. When abalone are infants, their sex is indeterminate so the entries in the first column are M, F, and I.

The variable names associated with the columns are shown in a Python list that gets named *names*. With the abalone data set, these names don't come from the first row of data, but from a separate file on the UC Irvine website. The first variable in the list is Sex—the sex of the animal. The last variable in the list is Rings. These are shell rings that are counted by slicing the shell and counting up the rings through a microscope. The number of rings is essentially the age of the abalone. The objective of the problem is to train a regression system to predict the Rings using easier, less time-consuming and less-expensive measurements.

Coding the Sex attribute is accomplished before the attribute matrix is normalized. The process is to build two columns to represent the three possible values. The logic of the construction is that the first column has a 1 if the corresponding row is from a male (M) and zero otherwise. The second column is 1 for female (F). Both columns are zero if the example is an infant (I). The new columns that replace Sex are given the names Sex1 and Sex2.

Once this coding is accomplished, then the attribute matrix contains all numeric values, and the example proceeds as in earlier examples. It normalizes the variables to zero mean and unit standard deviation, and then it applies the LARS algorithm introduced earlier to develop coefficient curves. The printed output shows the order in which variables enter into the solution of the penalized linear regression solution. You'll observe that both the two columns coding for Sex appear in the solution.

Figure 4-9 shows the coefficient curves that result from LARS applied to this problem. Chapter 5 delves more into performance, with different approaches to this problem.

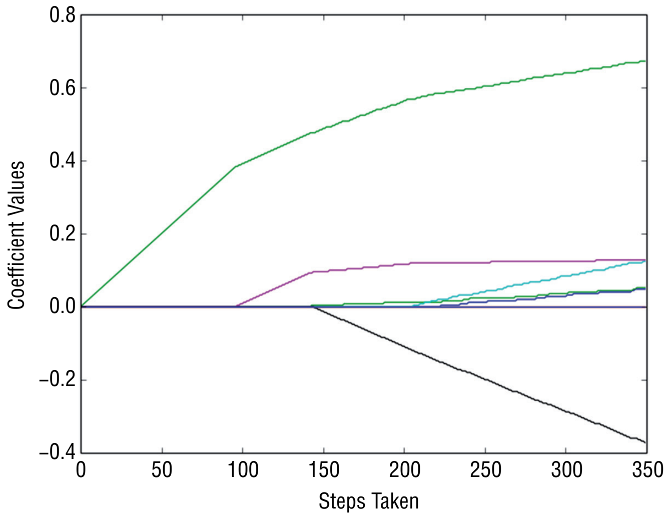

Figure 4-9: Coefficient curves for LARS trained on abalone data with coded categorical variable

This section discussed several extensions to penalized regression that broaden its utility to cover a wide class of problems. The section described a simple and frequently effective method of converting a classification problem to an ordinary regression problem. It also discussed how to convert a binary classifier into a multiclass classifier. The section went on to discuss how to model nonlinear behaviors using linear regression by adding new attributes that are nonlinear functions of the old attributes. Finally, the section showed how to turn categorical variables into real-valued variables so you can train linear algorithms on categorical variables. This method of converting categorical variables doesn't just work for linear regression. It is also useful for other linear methods such as support vector machines.

Summary

The goal of this chapter was to lay the groundwork for you to confidently understand and use the Python packages implementing the algorithms described here. The chapter described the nature of the input data set as a column vector of outcomes to be predicted and a table of attributes upon which to base the predictions. Chapter 3, the previous chapter, demonstrated that predictive models need to have their complexity tuned to get the best performance for a given problem complexity and data set size. Chapter 3 also showed some methods for introducing a tuning parameter into linear regression. This chapter built on that background and introduced several minimization problems where a tunable coefficient penalty was added to the error penalty from least squares regression. As was demonstrated, this tunable penalty on linear coefficient sizes results in suppression of the coefficients to a greater or lesser degree and thereby adds a complexity adjustment. You saw how to tune the complexity of the resulting models by using the error on out-of-sample data to achieve optimum performance.

The chapter described principles of operation for two modern algorithms for solving the penalized regression minimization problem and python code implementing the main features of the algorithms in order for you to have a concrete instantiation of the core of the algorithms to make the principals of operation clear. The plain regression problem (numeric features and numeric targets) served as the exemplar for in-depth coverage of algorithms. The chapter showed several extensions to broaden the use cases to include binary classification problems, multiclass classification problems, problems with nonlinear relationship between attributes and outcomes, and problems with non-numeric features.

The next chapter, Chapter 5, will use Python packages implementing these algorithms to run through a series of examples that were chosen to exercise a variety of different problem characteristics in order to cement these ideas. Based on what you've learned in this chapter, the various parameters and methods in the Python packages will make sense for you.

References

1. Bradley Efron, Trevor Hastie, Iain Johnstone, and Robert Tibshirani (2004). "Least Angle Regression." *Annals of Statistics*, *32*(2), 407–499.
2. Jerome H. Friedman, Trevor Hastie and Rob Tibshirani (2010). "Regularization Paths for Generalized Linear Models via Coordinate Descent." *Journal of Statistical Software*, vol. 33, issue 1, Feb 2010.