Overview

Here we’ll cover the remaining elements of logistic regression, including what happens when you call .fit to train the model, and the statistical assumptions you should be aware of when using this modeling technique. Delving into the details of gradient descent for model training opens up a discussion of the bias-variance tradeoff and the concepts of overfitting and underfitting. You will learn how to use L1 and L2 regularization with a logistic regression to prevent overfitting and how to use the practice of cross-validation to decide the regularization strength. After getting experience with these techniques on a synthetic data set, you’ll do a little feature engineering to try and improve performance of an underfit model with the case study data. After reading this chapter you should have a firm enough foundation of knowledge of logistic regression to use it in your work, and an understanding of how regularization can be used in the model fitting process to take advantage of the bias-variance tradeoff and improve model performance on unseen data.

Introduction

In this chapter, we will introduce the remaining details of logistic regression left over from the previous chapter. In addition to being able to use scikit-learn to fit logistic regression models, you will gain insight into the gradient descent procedure, which is similar to the processes that are used "under the hood" to accomplish model fitting. Finally, we'll complete our discussion of the basic logistic regression model by familiarizing ourselves with the formal statistical assumptions of this method.

We begin our exploration of the foundational machine learning concepts of overfitting, underfitting, and the bias-variance trade-off by examining how the logistic regression model can be extended to address the overfitting problem. After reviewing the mathematical details of the regularization methods that are used to alleviate overfitting, you will learn a useful practice for tuning the hyperparameters of regularization: cross-validation. Through the methods of regularization and some simple feature engineering, you will gain an understanding of how to improve both overfit and underfit models.

Although we’re focusing on logistic regression in this chapter, the concepts of overfitting, underfitting, regularization, and the bias-variance tradeoff are relevant to nearly all supervised modeling techniques in machine learning.

Estimating the Coefficients and Intercepts of Logistic Regression

In the previous chapter, we learned that the coefficients of a logistic regression model (each of which goes with a particular feature), as well as the intercept, are determined using the training data, when the .fit method is called on a logistic regression model in scikit-learn. These numbers are called the parameters of the model, and the process of finding the best values for them is called parameter estimation. Once the parameters are found, the logistic regression model is essentially a finished product; therefore, with just these numbers, we can use a logistic regression model in any environment where we can perform common mathematical functions.

It is clear that the process of parameter estimation is important, since this is how we can make a predictive model from our data. So, how does parameter estimation work? To understand this, the first step is to familiarize ourselves with the concept of a cost function. A cost function is a way of telling how far away the model predictions are from perfectly describing the data; that is to say, the larger the errors between the model predictions and the actual data, then the larger the "cost" returned by the cost function. This is a straightforward concept for regression problems: the difference between predictions and true values can be used for the cost, after going through a transformation (such as absolute value or squaring) which makes the value of the cost positive, and then averaging this over all the training samples.

For classification problems, especially in fitting logistic regression models, a typical cost function is the log-loss function, also called cross entropy loss. This is the cost function that scikit-learn uses, in a modified form, to fit logistic regression. Here is the definition of the log-loss function:



Figure 4.1: The log-loss function

Here, there are n training samples, yi is the true label (0 or 1) of the ith sample, pi is the predicted probability that the label of the ith sample equals 1, and log is the natural logarithm. The summation notation (that is, the uppercase Greek letter, Sigma) over all the training samples and division by n serve to take the average of this cost function over all training samples. With this in mind, take a look at the following graph of the natural logarithm function, and consider what the interpretation of this cost function is:

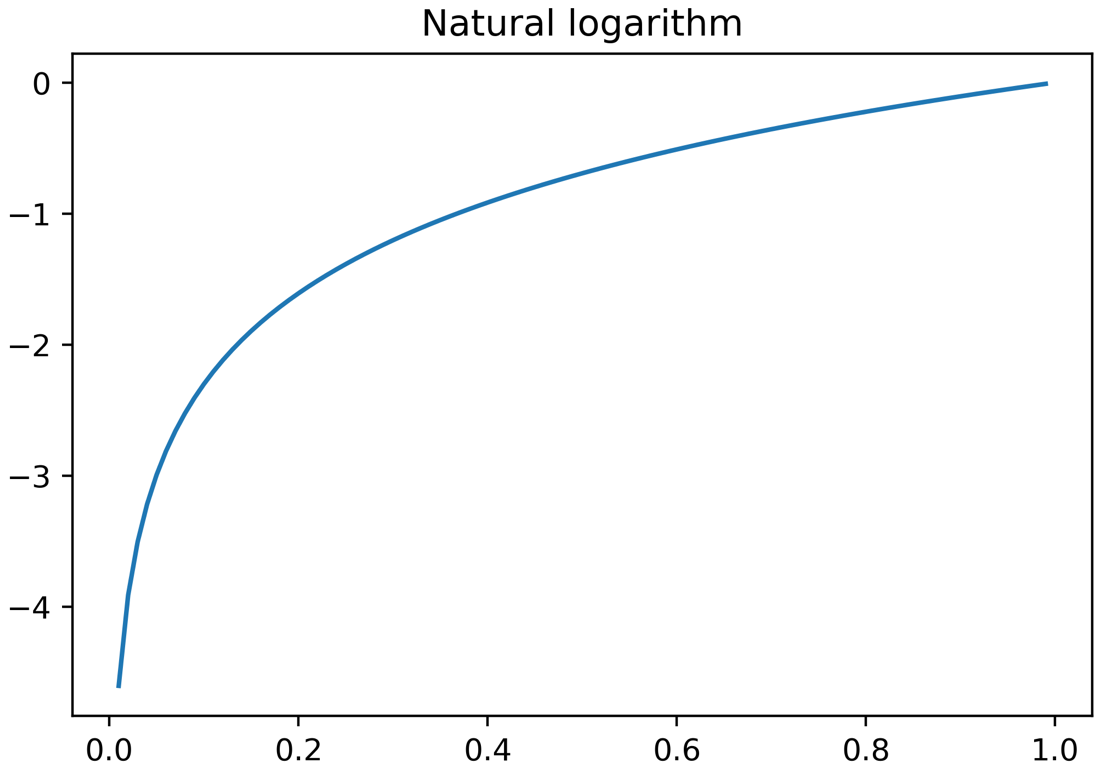


Figure 4.2: Natural logarithm on the interval (0, 1)

To see how the log-loss cost function works, consider its value for a sample where the true label is 1. y = 1 in this case, so the second part of the cost function, (1 - yi)log(1 - pi), will be exactly equal to 0 and will not affect the value. Then the value of the cost function is -yilog(pi) = -log(pi) since yi = 1. So, the cost for this sample is simply the negative of the natural logarithm of the predicted probability. Now since the true label for the sample is 1, consider how should the cost function behave. We expect that for predicted probabilities that are close to 1, the cost function will be small, representing a small error for predictions that are closer to the true value. For predictions that are closer to 0, it will be larger, since the cost function is supposed to take on larger values the more "wrong" the prediction is.

From the graph of the natural logarithm in Figure 4.1 we can see that for values of p that are closer to 0, the natural logarithm takes on increasingly negative values. This means the cost function will take on increasingly positive values, so that the cost of classifying a positive sample with a very low probability is relatively high, as it should be. Conversely, if the predicted probability is closer to 1, then the graph indicates the cost will be closer to 0 – again this is as expected for a prediction that is "more correct". Therefore, the cost function behaves as expected for a positive sample. A similar observation can be made for samples where the true label is 0.

Now we understand how the log-loss cost function works for logistic regression. But what does this have to do with how the coefficients and the intercept are determined? We will learn in the next section.

Gradient Descent to Find Optimal Parameter Values

The problem of finding the parameter values (coefficients and intercept) for a logistic regression model using a log-loss cost boils down to a problem of optimization: we would like to find the set of parameters that results in the minimum cost, since costs are higher for worse predictions. In other words, we want the set of parameters that is the "least wrong" on average over all of the training samples. This process is done for you automatically by the .fit method of the logistic regression model in scikit-learn. There are different solution techniques for finding the set of parameters with the lowest cost and you can choose which one you would like to use with the solver keyword when you are instantiating the model class. All of these methods work somewhat differently. However, they are all based on the concept of gradient descent.

The gradient descent process starts with an initial guess. The choice of the initial guess is not that important for logistic regression and you don't need to make it manually; this is handled by the solver keyword. However, for more advanced machine learning algorithms such as deep neural networks, selection of the initial guesses for parameters requires more attention.

For the sake of illustration, we will consider a problem where there is only one parameter to estimate. We'll look at the value of a hypothetical cost function (y = f(x) = x2 – 2x) and devise a gradient descent procedure to find the value of the parameter, x, for which the cost, y, is the lowest. Here, we choose some x values, create a function that returns the value of the cost function, and look at the value of the cost function over this range of parameters.

The code to do this is as follows:

X\_poly = np.linspace(-3,5,81)

print(X\_poly[:5], '...', X\_poly[-5:])

Here is the output of the print statement:

[-3. -2.9 -2.8 -2.7 -2.6] ... [4.6 4.7 4.8 4.9 5. ]

The remaining code snippet is as follows:

def cost\_function(X):

return X \* (X-2)

y\_poly = cost\_function(X\_poly)

plt.plot(X\_poly, y\_poly)

plt.xlabel('Parameter value')

plt.ylabel('Cost function')

plt.title('Error surface')

The resulting plot should appear as follows:

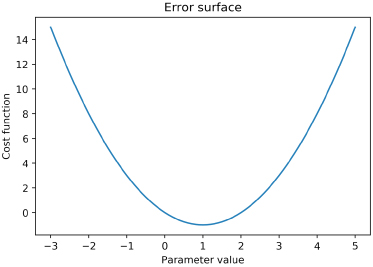


Figure 4.3: A cost function plot

Looking at the error surface in Figure 4.3, which is the plot of the cost function over a range of parameter values, it's pretty evident what parameter value will result in the lowest value of the cost function: x = 1. In fact, with some calculus, you could easily confirm this by setting the derivative equal to zero and then solving for x, confirming that x = 1 is the minimum. However, generally speaking, it is not always feasible to solve the problem so simply. In cases where it is necessary to use gradient descent, we don't always have knowledge of how the entire error surface looks. Rather, after we've chosen the initial guess for the parameter, all we're able to know is the direction of the error surface in the immediate vicinity of that point.

Gradient descent is an iterative algorithm; starting from the initial guess, we try to find a new guess that lowers the cost function and continue with this until we've found a good solution. We are trying to move "downhill" on the error surface, but we only know which direction to move in and how far to move in that direction, based on the shape of the error surface in the immediate neighborhood of our current guess. In mathematical terms, we only know the value of the derivative (which is called the gradient in more than one dimension) at the parameter value of the current guess. If you have not studied calculus, you can think of the gradient as telling you which direction is downhill, and how steep the hill is from where you're standing. We use this information to "take a step" in the direction of decreasing error. How big a step we decide to take, depends on the learning rate. Since the gradient declines toward the direction of decreasing error, we want to take a step in the direction that is the negative of the gradient.

These notions can be formalized in the following equation. To get to the new guess, xnew, from the current guess, xold, where f'(xold) is the derivative (that is, the gradient) of the cost function at the current guess:



Figure 4.4: Equation to obtain the new guess from the current guess

In the following figure, we can see the results of starting a gradient descent procedure from x = 4.5, with a learning rate of 0.75, and then optimizing x to attain the least value of the cost function:

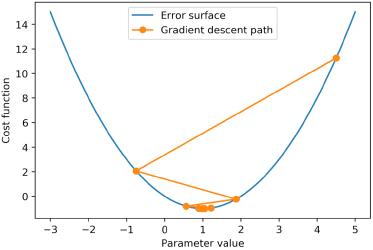


Figure 4.5: The gradient descent path

Gradient descent also works in higher-dimensional spaces; in other words, with more than one parameter. However, you can only visualize up to a two-dimensional error surface (that is, two parameters at a time on a three-dimensional plot) on a single graph.

Having described the workings of gradient descent, let's perform an exercise to implement the gradient descent algorithm, expanding on the example of this section.

Exercise 16: Using Gradient Descent to Minimize a Cost Function

In this exercise, our task is to find the best set of parameters in order to minimize the following hypothetical cost function: y = f(x) = x2 – 2x. To do this, we will employ gradient descent, which was described in the preceding section. Perform the following steps to complete the exercise:

Note

For Exercises 16 to18 and Activity 4, the code and the resulting output have been loaded in a Jupyter Notebook that can be found at ~~<http://bit.ly/2ZAy2Pr>~~. You can scroll to the appropriate section within the Jupyter Notebook to locate the exercise or activity of choice.

1. Create a function that returns the value of the cost function and look at the value of the cost function over a range of parameters. You can use the following code to do this (note this repeats code from the preceding section):

X\_poly = np.linspace(-3,5,81)

print(X\_poly[:5], '...', X\_poly[-5:])

def cost\_function(X):

return X \* (X-2)

y\_poly = cost\_function(X\_poly)

plt.plot(X\_poly, y\_poly)

plt.xlabel('Parameter value')

plt.ylabel('Cost function')

plt.title('Error surface')

You will obtain the following plot of the cost function:

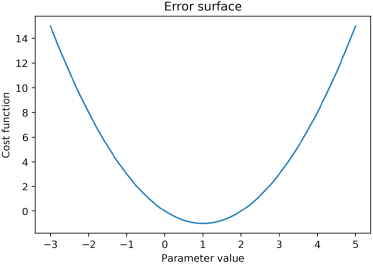


Figure 4.6: A cost function plot

1. Create a function for the value of the gradient. This is the analytical derivative of the cost function. Use this function to evaluate the gradient at the point x = 4.5, then use this in combination with the learning rate to find the next step of the gradient descent process.

def gradient(X):

return (2\*X) - 2

x\_start = 4.5

learning\_rate = 0.75

x\_next = x\_start - gradient(x\_start)\*learning\_rate

x\_next

Note

It doesn't matter if you haven't studied calculus and don't understand this part; you can just take it as a given that this is the function for the gradient. In some applications it's not actually possible to calculate an analytical derivative, so this may need to be numerically approximated.

After running the cell with x\_next, you will obtain the following output:

-0.75

This is the next gradient descent step after x = 4.5.

1. Plot the gradient descent path, from the starting point to the next point, using the following code:

plt.plot(X\_poly, y\_poly)

plt.plot([x\_start, x\_next], [cost\_function(x\_start), cost\_function(x\_next)], '-o')

plt.xlabel('Parameter value')

plt.ylabel('Cost function')

plt.legend(['Error surface', 'Gradient descent path'])

You will obtain the following output:

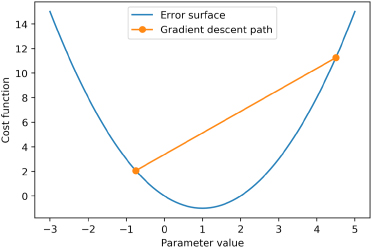


Figure 4.7: The first gradient descent path step

Here, it appears as though we've taken a step in the right direction. However, it's clear that we've "overshot" where we want to be. It may be that our learning rate is too large, and consequently, we are taking steps that are too big. While tuning the learning rate will be a good idea to converge toward an optimal solution more quickly, in this example, we can just continue illustrating the remainder of the process. Here, it looks like we may need to take a few more steps. In practice, gradient descent continues until the size of the steps become very small, or the change in the cost function becomes very small (you can specify how small by using the tol argument in the scikit-learn logistic regression), indicating that we're "close enough" to a good solution – that is, a local minimum of the cost function. For this example, we'll just take a total of 14 steps, or iterations, beyond the initial guess (note that you can also set the maximum number of iterations in scikit-learn with max\_iter).

1. Perform 14 iterations to converge toward the local minimum of the cost function by using the following code snippet (note that iterations = 15 but the endpoint is not included in the call to range()):

iterations = 15

x\_path = np.empty(iterations,)

x\_path[0] = x\_start

for iteration\_count in range(1,iterations):

derivative = gradient(x\_path[iteration\_count-1])

x\_path[iteration\_count] = x\_path[iteration\_count-1] - (derivative\*learning\_rate)

x\_path

You will obtain the following output:

array([ 4.5 , -0.75 , 1.875 , 0.5625 , 1.21875 ,

0.890625 , 1.0546875 , 0.97265625, 1.01367188, 0.99316406,

1.00341797, 0.99829102, 1.00085449, 0.99957275, 1.00021362])

This for loop stores the successive estimates in the array x\_path, using the current estimate to calculate the derivative and find the next estimate. From the resulting values of the gradient descent process, it looks like we've gotten very close (1.00021362) to the optimal solution (1).

1. Plot the gradient descent path using the following code:

plt.plot(X\_poly, y\_poly)

plt.plot(x\_path, cost\_function(x\_path), '-o')

plt.xlabel('Parameter value')

plt.ylabel('Cost function')

plt.legend(['Error surface', 'Gradient descent path'])

You will obtain the following output:

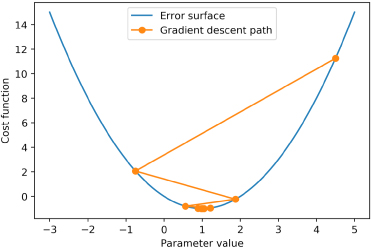


Figure 4.8: The gradient descent path

We encourage you to repeat the previous procedure with different learning rates in order to see how they affect the gradient descent path. With the right learning rate, it's possible to converge on a highly accurate solution very quickly. While the choice of learning rate can be important in different machine learning applications, for logistic regression, the problem is usually pretty easy to solve and you don't need to select a learning rate in scikit-learn.

As you experimented with different learning rates, did you notice what happened when the learning rate was greater than one? In this case, the step that we take in the direction of the decreasing error is too large and we actually wind up with a higher error. This problem can compound itself and actually lead the gradient descent process away from the region of minimum error. On the other hand, if the step size is too small, it can take a very long time to find the desired solution.

Assumptions of Logistic Regression

Since it is a classical statistical model, similar to the F-test and Pearson correlation we already examined, logistic regression makes certain assumptions about the data. While it's not necessary to follow every one of these assumptions in the strictest possible sense, it's good to be aware of them. That way, if a logistic regression model is not performing very well, you can try to investigate and figure out why, using your knowledge of the ideal situation in which a logistic regression would work well. You may find slightly different lists of the specific assumptions from different resources, however those that are listed here are widely accepted.

Features Are Linear in the Log Odds

We learned about this assumption in the previous chapter, Chapter 3, Details of Logistic Regression and Feature Exploration. Logistic regression is a linear model, so it will only work well as long as the features are effective at describing a linear trend in the log odds. In particular, logistic regression won't capture interactions, polynomial features, or the discretization of features, on its own. You can, however, specify all of these as "new features" – even though they are engineered from existing features.

Remember from the previous chapter that the most important feature from univariate feature exploration, PAY\_1, was not found to be linear in the log odds.

No Multicollinearity of Features

Multicollinearity means that features are correlated with each other. The worst violation of this assumption is when features are perfectly correlated with each other, such as one feature being identical to another, or when one feature equals another multiplied by a constant. We can investigate the correlation of features using the correlation plot that we're already familiar with from univariate feature selection. Here, we load the data and recreate the plot, in the same way that we did previously:

Note

Adjust the path in the following code to the location where you saved the cleaned data from Chapter 1, Data Exploration and Cleaning.

df = pd.read\_csv('../Data/Chapter\_1\_cleaned\_data.csv')

features\_response = df.columns.tolist()

items\_to\_remove = ['ID', 'SEX', 'PAY\_2', 'PAY\_3', 'PAY\_4', 'PAY\_5',

'PAY\_6', 'EDUCATION\_CAT', 'graduate school',

'high school', 'none','others', 'university']

features\_response = [item for item in features\_response

if item not in items\_to\_remove]

corr = df[features\_response].corr()

mpl.rcParams['figure.dpi'] = 400 #high res figures

sns.heatmap(corr,

xticklabels=corr.columns.values,

yticklabels=corr.columns.values,

center=0)

The resulting plot is as follows:

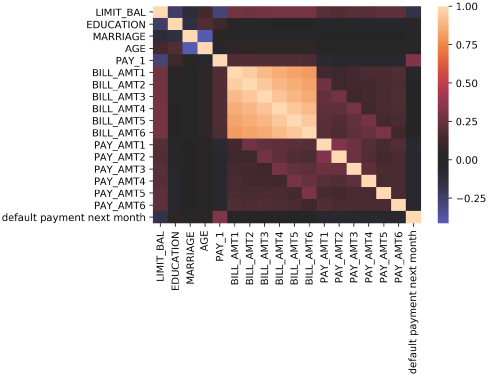


Figure 4.9: A correlation plot of features and the response

We can see from the correlation plot what perfect correlation looks like: since every feature and the response variable has a correlation of 1 with itself, we can see that a correlation of 1 is a light, cream color. From the color bar, which doesn't include -1, we know there are no correlations with that value.

The clearest examples of correlated predictors in our case study data are the BILL\_AMT features. It makes intuitive sense that bills might be similar from month to month for a given account. For instance, there may be an account that typically carries a balance of zero, or an account that has a large balance that is taking a while to pay off. Are any of the BILL\_AMT features perfectly correlated? The answer is no, as we can confirm from Figure 4.9. So, while these features may not contribute much independent information, we won’t remove them at this point.

The Independence of Observations

This is a common assumption in classical statistical models, including linear regression. Here, the observations (or samples) are assumed to be independent. Does this make sense with the case study data? We'd want to confirm with our client whether the same individual can hold multiple credit accounts across the dataset and consider what to do depending on how common it was. Let's assume we've been told that in our data each credit account belongs to a unique person, so we may assume independence of observations in this respect.

Across different domains of data, some common violations of independence of observations are as follows:

* Spatial autocorrelation of observations; for example, in natural phenomena such as soil types, where observations that are geographically closer to each other may be similar to each other.
* Temporal autocorrelation of observations, which may occur in time series data. Observations at the current point in time are usually assumed to be correlated to the most recent point(s) in time.

However, these issues are not relevant to our case study data.

No Outliers

Outliers are observations where the value of the feature(s) or response are very far from most of the data, or are different in some other way. A more appropriate term for an outlier observation of a feature value is a high leverage point, as the term "outlier" is usually applied to the response variable. However, in our binary classification problem, it's not possible to have an outlier value of the response variable, since it can only take on the values 0 and 1. In practice, you may see both of these terms used to refer to features.

To see why these kinds of points can have an adverse effect on linear models in general, take a look at this synthetic linear data with 100 points and the line of best fit that results from linear regression:

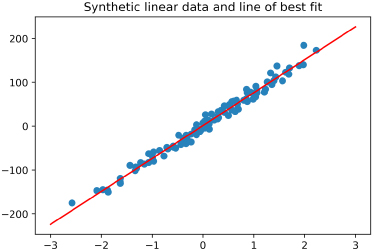


Figure 4.10: "Well-behaved" linear data and a regression fit

Here, the model intuitively appears to be a good fit for the data. However, what if an outlier feature value is added? In this case, we add a point with an x value that is very different from most of the observations and a y value that is in a similar range to the other observations. We then show the resulting regression line:

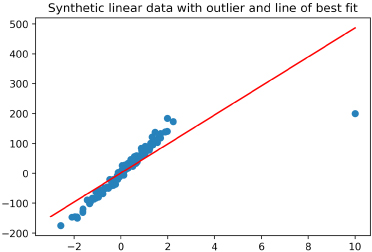


Figure 4.11: A plot showing what happens when an outlier is included

Due to the presence of a single high leverage point, the regression model fit for all the data is no longer a very good representation of much of the data. This shows the potential effect of just a single data point on linear models, especially if that point doesn't appear to follow the same trend as the rest of the data.

There are methods to deal with outliers. But a more fundamental question to ask is "Is data like this realistic?". If the data doesn't seem right, it is a good idea to ask the client whether the outliers are believable. If not, they should be excluded. However, if they do represent valid data, then non-linear models or other methods should be used.

With our case study data, we did not observe outliers in the histograms that we plotted during feature exploration. Therefore, we don't have this concern.

How Many Features Should You Include?

This is not so much an assumption as it is guidance on model building. There is no clear-cut law that states how many features to include in a logistic regression model. However, a common rule of thumb is the "rule of 10," which states that for every 10 occurrences of the rarest outcome class, 1 feature may be added to the model. So, for example, in a binary logistic regression problem with 100 samples, if the class balance has 20% positive outcomes and 80% negative outcomes, then there are only 20 positive outcomes in total, and so only 2 features should be used in the model. A "rule of 20" has also been suggested, which would be a more stringent limit on the number of features to include (1 feature in our example).

Another point to consider in the case of binary features, such as those that result from one-hot encoding, is how many samples will have a positive value for that feature. If the feature is very imbalanced, in other words, with very few samples containing either a 1 or a 0, it may not make sense to include it in the model.

For the case study data, we are fortunate to have a relatively large number of samples and relatively balanced features, so these are not concerns.

The Motivation for Regularization: The Bias-Variance Trade-off

We can extend the basic logistic regression model that we have learned about by using a powerful concept known as shrinkage or regularization. In fact, every logistic regression that you have fit so far in scikit-learn has used some amount of regularization. That is because it is a default option in the logistic regression model object. However, until now, we have ignored it.

As you learn about these concepts in greater depth, you will also become familiar with a few foundational concepts in machine learning: overfitting, underfitting, and the bias-variance trade-off. A model is said to overfit the training data if the performance of the model on the training data (for example, the ROC AUC) is substantially better than the performance on a held-out test set. In other words, good performance on the training set does not generalize to the unseen test set. We started to discuss these concepts in Chapter 2, Introduction to Scikit-Learn and Model Evaluation, when we distinguished between model training and testing scores.

When a model is overfit to the training data, it is said to have high variance. In other words, whatever variability exists in the training data, the model has learned this very well – in fact, too well. This will be reflected in a high model training score. However, when such a model is used to make predictions on new and unseen data, the performance is lower. Overfitting is more likely in the following circumstances:

1. There are a large number of features available in relation to the number of samples. In particular, there may be so many possible features that it is cumbersome to directly inspect all of them, like we were able to do with the case study data.
2. A complex model, that is, more complex than logistic regression, is used. These include models such as random forests, other ensemble models, or neural networks.

Under these circumstances, the model has an opportunity develop more complex hypotheses about the relationships between features and the response variable in the training data during model fitting, making overfitting more likely.

In contrast, if a model is not fitting the training data very well, this is known as underfitting, and the model is said to have high bias.

We can examine the differences between underfitting, overfitting, and the ideal that sits in between, by fitting polynomial models on some hypothetical data:

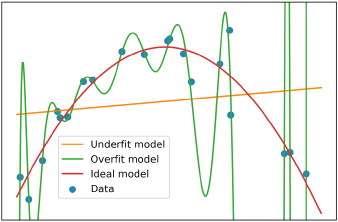


Figure 4.12: Quadratic data with underfit, overfit, and ideal models

In Figure 4.12 we can see that including too few features, in this case a linear model of y with just two features, a slope and intercept, is clearly not a good representation of the data. This is known as an underfit model. However, if we include too many features, that is, with many high-degree polynomial terms, such as x2, x3, x4,… x10, we can fit the training data almost perfectly. However, this is not necessarily a good thing. When we look at the results of the overfit model in between the training data points, where new predictions may need to be made, we can see that the model is unstable and may not provide reliable predictions for data that was not in the training set. We can tell this just based on an intuitive understanding of the relationship between the features and the response variable, which we can get from visualizing the data.

The synthetic data for this example was generated by a second-degree (that is, quadratic) polynomial. Knowing this, we could easily find the ideal model by fitting a second-degree polynomial to the training data, as shown in Figure 4.12. In general, however, we won't know what the ideal model formulation is ahead of time. For this reason, we need to compare training and testing scores to assess whether a model may be overfitting or underfitting.

In some cases, it may be desirable to introduce some bias into the model training process, especially if this decreases overfitting and increases model performance on new, unseen data. In this way, it may be possible to leverage the bias-variance trade-off to improve a model. We can use regularization methods to accomplish this. Additionally, we may also be able to use these methods for variable selection as part of the modeling process. Using a predictive model to select variables is an alternative to the univariate feature selection methods that we've already explored. We begin to experiment with these concepts in the following exercise.

Exercise 17: Generating and Modeling Synthetic Classification Data

In this exercise, we'll observe overfitting in practice by using a synthetic dataset. Consider yourself in the situation of having been given a binary classification dataset with many candidate features (200), where you don't have time to look through all of them individually. It's possible that some of these features are highly correlated or related in some other way. However, with this many variables, it can be difficult to effectively explore all of them. Additionally, the dataset has relatively few samples: only 1,000. We are going to generate this challenging dataset by using a feature of scikit-learn that allows you to create synthetic datasets for making conceptual explorations such as this. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook. They can be found at ~~<http://bit.ly/2ZAy2Pr>~~.

1. Import the make\_classification, train\_test\_split, LogisticRegression, and roc\_auc\_score classes using the following code:

from sklearn.datasets import make\_classification

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import roc\_auc\_score

Notice that we've imported several familiar classes from scikit-learn, in addition to a new one that we haven't seen before: make\_classification. This class does just what its name indicates – it makes data for a classification problem. Using the various keyword arguments, you can specify how many samples and features to include, and how many classes the response variable will have. There is also a range of other options that effectively control how "easy" the problem will be to solve.

Note

For more information, refer to <https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html>. Suffice to say that we've selected options here that make a reasonably easy-to-solve problem, with some curveballs thrown in. In other words, we expect high model performance, but we'll have to work a little bit to get it.

1. Generate a dataset with two variables, x\_synthetic and y\_synthetic. x\_synthetic has the 200 candidate features and y\_synthetic the response variable, each for 1,000 samples. Use the following code:

X\_synthetic, y\_synthetic = \

make\_classification(n\_samples=1000, n\_features=200, n\_informative=3, n\_redundant=10, n\_repeated=0, n\_classes=2, n\_clusters\_per\_class=2, weights=None, flip\_y=0.01, class\_sep=0.8, hypercube=True, shift=0.0, scale=1.0, shuffle=True, random\_state=24)

1. Examine the shape of the dataset and the class fraction of the response variable using the following code:

print(X\_synthetic.shape, y\_synthetic.shape)

print(np.mean(y\_synthetic))

You will obtain the following output:

(1000, 200) (1000,)

0.501

After checking the shape of the output, note that we've generated an almost perfectly-balanced dataset: close to a 50/50 class balance. It is also important to note that we've generated all the features so that they have the same shift and scale – that is, a mean of 0 with a standard deviation of 1. Making sure that the features are on the same scale, or have roughly the same range of values, is a key point for using regularization methods – and we'll see why later. If the features in a raw dataset are on widely different scales, it is advisable to normalize them so that they are on the same scale. Scikit-learn has functionality to make this easy, which we'll learn about in the activity at the end of this chapter.

1. Plot the first few features as histograms to show that the range of values is the same using the following code:

for plot\_index in range(4):

plt.subplot(2, 2, plot\_index+1)

plt.hist(X\_synthetic[:, plot\_index])

plt.title('Histogram for feature {}'.format(plot\_index+1))

plt.tight\_layout()

You will obtain the following output:

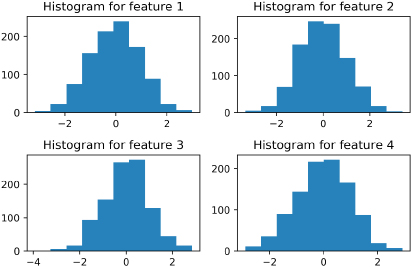


Figure 4.13: Histograms for the first 4 of 200 synthetic features

Because we generated this dataset, we don't need to directly examine all 200 features to make sure that they're on the same scale. So, what are the possible concerns with this dataset? The data are balanced in terms of the class fractions of the response variable, so we don't need to undersample, oversample, or use other methods that are helpful for imbalanced data. What about relationships among the features themselves, and the features and response variable? There are a lot of these relationships and it is a challenge to investigate them all directly. Based on our "rule of thumb", 200 features is too many (especially going by the "rule of 10"). We have 500 observations in the rarest class, so by that rule we shouldn't have more than 50 features. It's possible that with so many features the model training procedure will overfit. We will now start to learn how to use options in the scikit-learn logistic regression to prevent this.

1. Split the data into training and testing sets using an 80/20 split, and then instantiate a logistic regression model object using the following code:

X\_syn\_train, X\_syn\_test, y\_syn\_train, y\_syn\_test = \

train\_test\_split(

X\_synthetic, y\_synthetic,

test\_size=0.2, random\_state=24)

lr\_syn = \

LogisticRegression(solver='liblinear', penalty='l1', C=1000, random\_state=1)

lr\_syn.fit(X\_syn\_train, y\_syn\_train)

Notice here that we are specifying some new options in the logistic regression model, which so far, we have not paid attention to. First, we specify the penalty argument to be l1. This means we are going to use L1 regularization, which is also known as lasso regularization. We'll discuss the mathematical definition of this shortly. Second, notice that we have set the C parameter to be equal to 1,000. C is the "inverse of regularization strength," according to the scikit-learn documentation (<https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html>). This means that higher values of C correspond to less regularization. By choosing a relatively large number such as 1,000, we are using relatively little regularization. The default value of C is 1. So, we are not really using much regularization here, rather, we are simply becoming familiar with the options to do so. Finally, we are using the liblinear solver, which we have used in the past. Although we happen to be using scaled data here (all features have a mean of 0 and standard deviation of 1), it's worth noting at this point that among the various options we have available for solvers, liblinear is "robust to unscaled data." Also note that liblinear is one of only two solver options that support the L1 penalty – the other option being saga.

Note

You can find out more information on available solvers at https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression.

1. Fit the logistic regression model on the training data using the following code:

lr\_syn.fit(X\_syn\_train, y\_syn\_train)

Here is the output:

LogisticRegression(C=1000, penalty='l1', random\_state=1, solver='liblinear')

1. Calculate the training score using this code, by first getting predicted probabilities, then obtaining the ROC AUC:

y\_syn\_train\_predict\_proba = lr\_syn.predict\_proba(X\_syn\_train)

roc\_auc\_score(y\_syn\_train, y\_syn\_train\_predict\_proba[:,1])

The output should be:

0.9420000000000001

1. Calculate the testing score similar to how the training score was obtained:

y\_syn\_test\_predict\_proba = lr\_syn.predict\_proba(X\_syn\_test)

roc\_auc\_score(y\_syn\_test, y\_syn\_test\_predict\_proba[:,1])

The output should be:

0.8075807580758075

From these results, it's apparent that the logistic regression model has overfit the data. That is, the ROC AUC score on the training data is substantially higher than that of the testing data.

Lasso (L1) and Ridge (L2) Regularization

Before applying regularization to a logistic regression model, let's take a moment to understand what regularization is and how it works. The two ways of regularizing logistic regression models in scikit-learn are called lasso (also known as L1 regularization) and ridge (also known as L2 regularization). When instantiating the model object from the scikit-learn class, you can choose penalty = 'l1' or 'l2'. These are called "penalties" because the effect of regularization is to add a penalty, or a cost, for having larger values of the coefficients in a fitted logistic regression model.

As we've already learned, coefficients in a logistic regression model describe the relationship between the log-odds of the response and each of the features. Therefore, if a coefficient value is particularly large, then a small change in that feature will have a large effect on the prediction. When a model is being fit, and is learning the relationship between features and the response variable, the model can start to learn the noise in the data. We saw this previously in Figure 4.12: if there are many features available when fitting a model, and there are no "guardrails" on the values that their coefficients can take, then the model fitting process may try to discover relationships between the features and the response variable that won't generalize to new data. In this way the model becomes tuned to the unpredictable, random noise that accompanies real-world, imperfect data. Unfortunately, this only serves to increase the model's skill at predicting the training data, which is not our ultimate goal. Therefore, we should seek to root out such spurious relationships from the model.

Lasso and ridge regularization use different mathematical formulations to accomplish this goal. These methods work by making changes to the cost function that is used for model fitting, which we introduced previously as the log-loss function. Lasso regularization uses what is called the 1-norm (hence the term L1):



Figure 4.14: Log-loss equation with lasso penalty

The 1-norm, which is the first term in the equation in Figure 4.14, is just the sum of the absolute values of the coefficients of the m different features. The absolute value is used because having a coefficient be large in either the positive or negative directions can contribute to overfitting. So, what else is different about this cost function from the log-loss function that we saw earlier? Well, now there is a C factor that is multiplied by the fraction in front of the sum of the log-loss function. This is the "inverse of regularization strength" as described in the scikit-learn documentation (<https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html>). Since this factor is in front of the term of the cost function that calculates the prediction error, as opposed to the term that does regularization, then making it larger makes the prediction error more important in the cost function, while regularization is made less important. In short, larger values of C lead to less regularization in the scikit-learn implementation.

L2, or ridge regularization, is similar to L1, except that instead of the sum of absolute values of coefficients, ridge uses the sum of their squares, called the 2-norm:

Diagram, schematic

Description automatically generated

Figure 4.15: Log-loss equation with ridge penalty

Note that if you look at the cost functions for logistic regression in the scikit-learn documentation, the specific form is different than what is used here, but the overall idea is similar. Additionally, after you become comfortable with the concepts of lasso and ridge penalties, you should be aware that there is an additional regularization method called elastic-net, which is a combination of both the lasso and ridge.

Why Are There Two Different Formulations of Regularization?

It may be that one or the other will provide better results for you, so you may wish to test them both. There is another key difference in these methods: the L1 penalty also performs feature selection, in addition to regularization. It does this by setting some coefficient values to exactly zero during the regularization process, effectively removing features from the model. L2 regularization makes the coefficient values smaller but does not completely eliminate them. Not all solver options in scikit-learn support both L1 and L2 regularization, so you will need to select an appropriate solver for the regularization technique you want to use.

Note

While the mathematical details of why this happens are beyond the scope of this book, for a more thorough explanation of this topic and further reading in general, we recommend the very readable (and free) resource, An Introduction to Statistical Learning by Gareth James, et al. In particular, see page 222 of the corrected 7th printing, for a helpful graphic on the difference between L1 and L2 regularization.

Intercepts and Regularization

We have not discussed intercepts very much, other than to note that we have been estimating them with our linear models, along with the coefficients that go with each feature. So, should you use an intercept? The answer is probably yes, until you've developed an advanced understanding of linear models and are certain that in a specific case you should not. However, such cases do exist, for example, in a linear regression where the features and the response variable have all been normalized to have a mean of zero.

Intercepts don't go with any particular feature. Therefore, it doesn't make much sense to regularize them, as they shouldn't contribute to overfitting. Notice that in the regularization penalty term for L1, the summation starts with j = 1, and similarly for L2: we have skipped σ0, which is the intercept term.

This is the ideal situation: not regularizing the intercept. However, because of the way that the different solvers are implemented in scikit-learn, the liblinear solver actually does this. There is, however, an intercept\_scaling option that you can supply to the model class to counteract this effect. We have not illustrated this here as, although it is theoretically incorrect, regularizing the intercept often does not have much effect on the model's predictive quality in practice.

Scaling and Regularization

As noted in the previous exercise, it is best practice to scale the data so that all the features have roughly the same range of values before using regularization. This is because the coefficients are all going to be subject to the same penalty in the cost function. If the range of values for a particular feature, such as LIMIT\_BAL in our dataset, is much larger than other features, such as PAY\_1, it may, in fact, be desirable to have a larger value for the coefficient of PAY\_1 and a smaller value for that of LIMIT\_BAL in order to put their effects on the same scale in the linear combination of features and coefficients that are used for model prediction. Normalizing all the features before using regularization avoids complications like this that arise simply from differences in scale.

In fact, scaling your data may also be necessary, depending on which solver you are using. The different variations on the gradient descent process available in scikit-learn may or may not be able to work effectively with unscaled data.

The Importance of Selecting the Right Solver

As we've come to learn, the different solvers available for logistic regression in scikit-learn have different behaviors regarding the following:

* Whether they support both L1 and L2 regularization
* How they treat the intercept during regularization
* How they deal with unscaled data

Note

There are other differences as well. A helpful table comparing these and other traits is available at https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression. You can use this table to decide which solver is appropriate for your problem.

To summarize this section, we have learned the mathematical foundations of lasso and ridge regularization. These methods work by shrinking the coefficient values toward 0, and in the case of the lasso, setting some coefficients to exactly 0 and thus performing feature selection. You can imagine that in our example of overfitting in Figure 4.12, if the complex, overfit model had some coefficients shrunk toward 0, it would look more like the ideal model, which has fewer coefficients.

Here is a plot of a regularized regression model, using the same high-degree polynomial features as the overfit model, but with a ridge penalty:

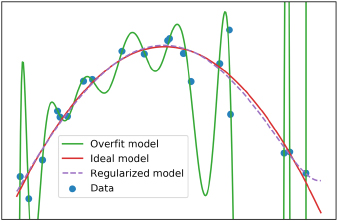


Figure 4.16: An overfit model and regularized model using the same features

The regularized model looks similar to the ideal model, demonstrating the ability of regularization to correct overfitting. Note, however, that the regularized model should not be recommended for extrapolation. Here, we can see that the regularized model starts to increase toward the right side of Figure 4.16. This increase should be viewed with suspicion, as there is nothing in the training data that makes it clear that this would be expected. This is an example of the general view that the extrapolation of model predictions outside the range of training data is not recommended. However, it is clear from Figure 4.16 that even if we didn't have knowledge of the model that was used to generate this synthetic data (as we typically don't have knowledge of the data-generating process in real-world predictive modeling work), we can still use regularization to reduce the effect of overfitting when a large number of candidate features are available.

Models and Feature Selection

L1 regularization is one way to use a model, such as logistic regression, to perform feature selection. Other methods include forward or backward stepwise selection from the pool of candidate features. Here is the high-level idea behind these methods: in the case of forward selection, features are added to the model one at a time, and the out-of-sample testing performance is observed along the way. At each iteration, the addition of all possible features from the candidate pool is considered, and the one resulting in the greatest increase in the out-of-sample performance is chosen. When adding additional features ceases to improve the model's performance, no more features need to be added from the candidates. In the case of backward selection, you first start with all the features in the model and determine which one you should remove: the one resulting in the smallest decrease in the out-of-sample testing performance. You can continue removing features in this way until the performance begins to decrease appreciably.

Cross Validation: Choosing the Regularization Parameter and Other Hyperparameters

By now, you may suspect that we could use regularization in order to decrease the overfitting we observed when we tried to model the synthetic data in Exercise 17, Generating and modeling Synthetic Classification Data. The question is, how do we choose the regularization parameter, C? C is an example of a model hyperparameter. Hyperparameters are different from the parameters that are estimated when a model is trained, such as the coefficients and the intercept of a logistic regression. Rather than being estimated by an automated procedure like the parameters are, hyperparameters are input directly by the user as keyword arguments, typically when instantiating the model class. So, how do we know what values to choose?

Hyperparameters are more difficult to estimate than parameters. This is because it is up to the data scientist to determine what the best value is, as opposed to letting an optimization algorithm find it. However, it is possible to programmatically choose hyperparameter values, which could arguably be viewed as a kind of optimization procedure in its own right. Practically speaking, in the case of the regularization parameter C, for example, this is most commonly done by fitting the model on one set of data with a particular value of C, determining model training performance, and then assessing the out-of-sample performance on another set of data.

We are already familiar with the concept of using model training and testing sets. However, there is a key difference here; for instance, what would happen if we were to use the testing set multiple times in order to see the effect of different values of C?

It may occur to you, that after the first time you use the unseen test set to assess the out-of-sample performance for a particular value of C, it is no longer an "unseen test set." While only the training data was used for estimating the model parameters (that is, the coefficients and the intercept), now the testing data is being used to estimate the hyperparameter C. Effectively, the testing data has now become additional training data, in the sense that it is being used to find a good value for the hyperparameter.

For this reason, it is common to divide the data into three parts: a training set, a testing set, and a validation set. The validation set serves multiple purposes:

Estimating Hyperparameters

The validation set can be repeatedly used to assess the out-of-sample performance with different hyperparameter values, to select hyperparameters.

A Comparison of Different Models

In addition to finding hyperparameter values for a model, the validation set can be used to determine the out-of-sample performance of different models; for example, if we wanted to compare logistic regression to random forest.

Note

Data Management Best Practices

As a data scientist, it's up to you to figure out how to divide up your data for different predictive modeling tasks. In the ideal case, you should reserve a portion of your data for the very end of the process, after you've already selected model hyperparameters and also selected the best model. This unseen test set is reserved for the last step, when it can be used to assess the endpoint of your model building efforts, to see how the final model generalizes to new unseen data. When reserving the testing set, it is good practice to make sure that the features and response have similar characteristics to the rest of the data. In other words, the class fraction should be the same, and the distribution of features should be similar. This way, the testing data should be representative of the data you build the model with.

While model validation is a good practice, it raises the question of whether the particular split we choose for the training, validation, and testing data has any effect on the outcomes that we are tracking. For example, perhaps the relationship between the features and the response variable is slightly different in the unseen test set that we have reserved, or in the validation set, versus the training set. It is likely impossible to eliminate all such variability, but we can use the method of cross-validation to avoid placing too much faith in one particular split of the data.

Scikit-learn provides convenient functions to facilitate cross-validation analyses. These functions play a similar role to train\_test\_split, which we have already been using, although the default behavior is somewhat different. Let's get familiar with them now; first, import these two classes:

from sklearn.model\_selection import StratifiedKFold

from sklearn.model\_selection import KFold

Similar to train\_test\_split, we need to specify what proportion of the dataset we would like to use for training versus testing. However, with cross-validation (specifically the k folds cross-validation that was implemented in the classes we just imported), rather than specifying a proportion directly we simply indicate how many folds we would like – that is, the "k folds." The idea here is that the data will be divided into k equal proportions. For example, if we specify four folds, then each fold will have 25% of the data. These folds will be the testing data in four separate instances of model training, while the remaining 75% from each fold will be used to train the model. In this procedure, each data point gets used as training data a total of k - 1 times, and as testing data only once.

When instantiating the class, we indicate the number of folds, whether or not to shuffle the data before splitting, and a random seed if we want repeatable results across different runs:

n\_folds = 4

k\_folds = KFold(n\_splits=n\_folds, shuffle=False)

Here, we've instantiated an object with four folds and no shuffling. The way in which we use the object that is returned, which we've called k\_folds, is by passing the features and response data that we wish to use for cross-validation, to the .split method of this object. This outputs an iterator, which means that we can loop through the output to get the different splits of training and testing data. If we took the training data from our synthetic classification problem, X\_syn\_train and y\_syn\_train, we could loop through the splits like this:

for train\_index, test\_index in k\_folds\_iterator.split(X\_syn\_train, y\_syn\_train):

The iterator will return the row indices of X\_syn\_train and y\_syn\_train, which we can use to index the data. Inside this for loop, we can write code to use these indices to select data for repeatedly training and testing a model object with different subsets of the data. In this way, we can get a robust indication of the out-of-sample performance when using one particular hyperparameter value, and then repeat the whole process using another hyperparameter value. Consequently, the cross-validation loop may sit nested inside an outer loop over different hyperparameter values. We'll illustrate this in the following exercise.

First though, what do these splits look like? If we were to simply plot the indices from train\_index and test\_index as different colors, we would get something that looks like this:

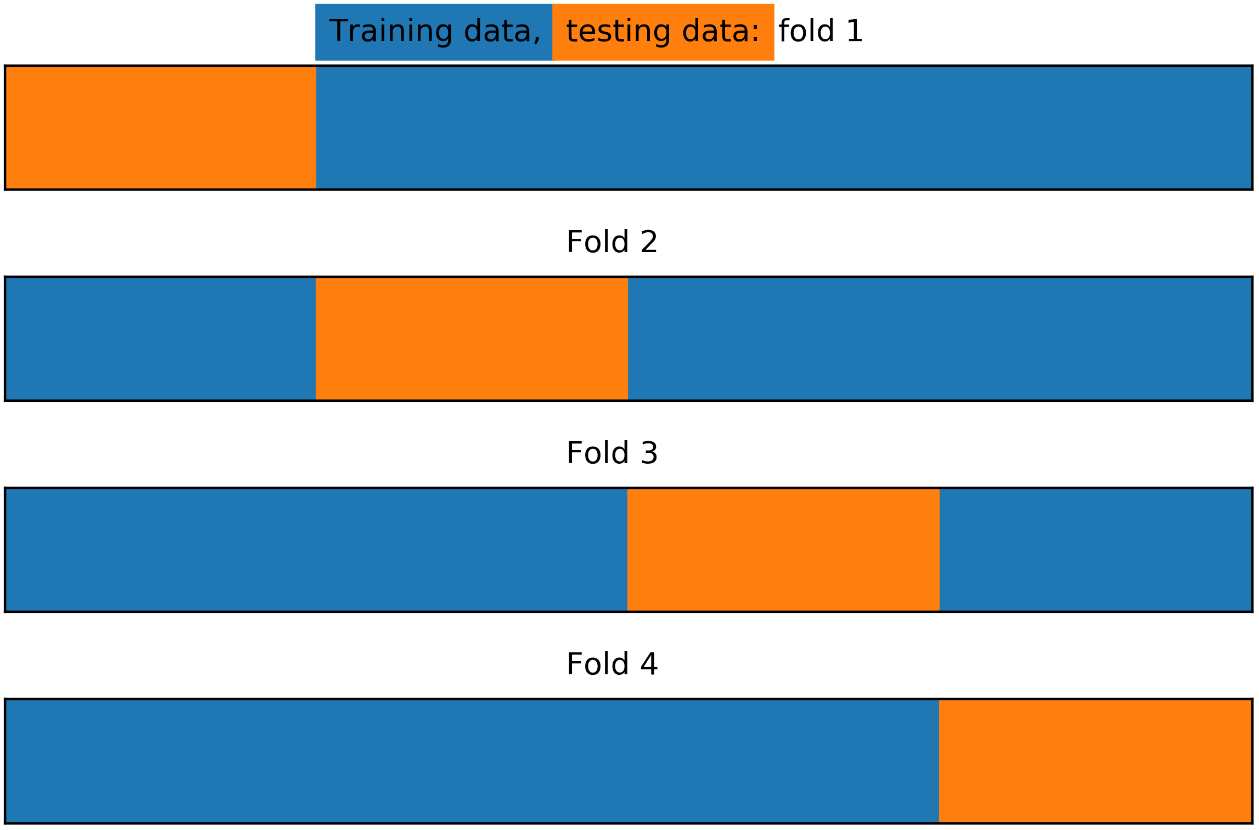


Figure 4.17: Train/test splits for k-folds with four folds and no shuffling

Here, we see that with the options we've indicated for the KFold class, the procedure has simply taken the first 25% of the data, according to the order of rows, as the first testing fold, then the next 25% of data for the second fold, and so on. But what if we wanted stratified folds? In other words, what if we wanted to ensure that the class fractions of the response variable were equal in every fold? While train\_test\_split allows for this option as a keyword argument, there is a separate StratifiedKFold class that implements this for cross-validation. We can illustrate how the stratified splits will look as follows:

k\_folds = StratifiedKFold(n\_splits=n\_folds, shuffle=False)

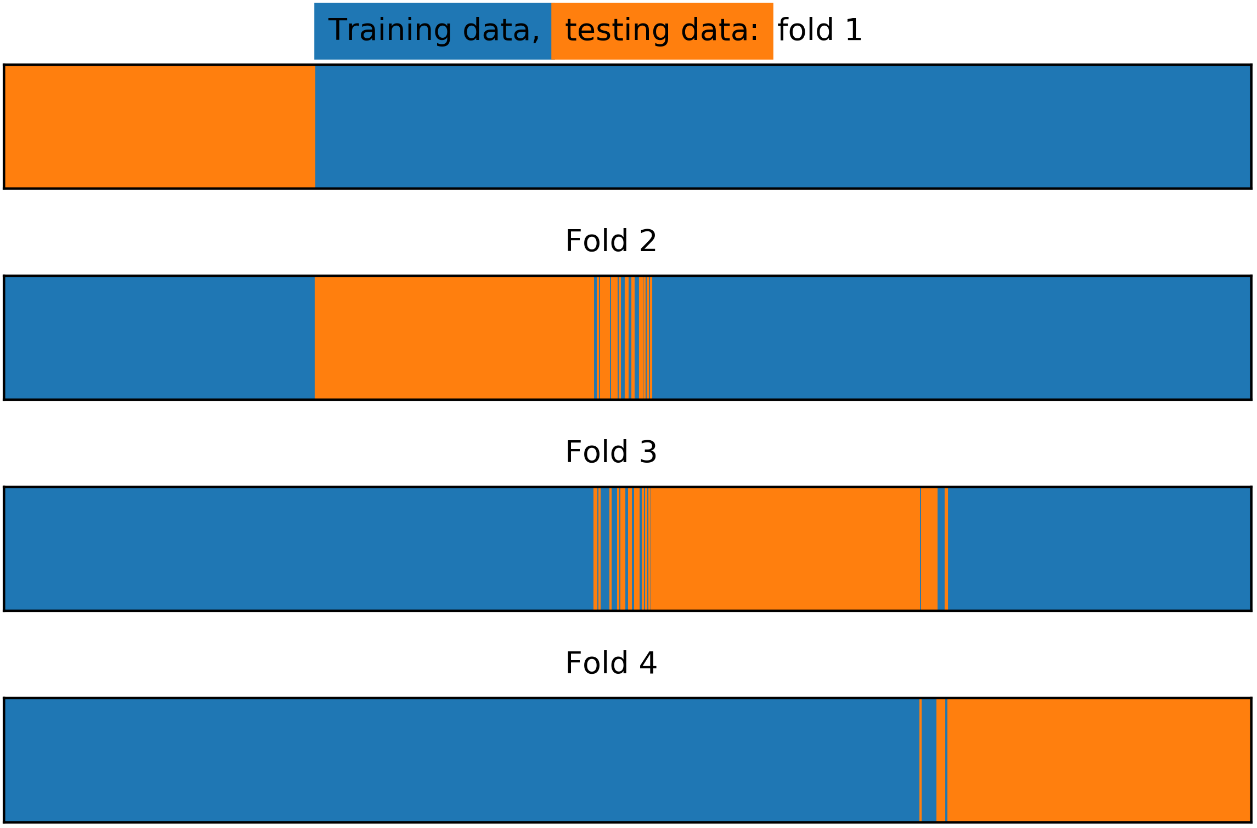


Figure 4.18: Train/test splits for stratified k-folds

In Figure 4.18 we can see that there has been some amount of "shuffling" between the different folds. The procedure has moved samples between folds as necessary, to ensure that the class fractions in each fold are equal.

Now, what if we want to shuffle the data to choose samples from throughout the range of indices for each testing fold? First, why might we want to do this? Well, with the synthetic that we've created for our problem, we can be certain that the data is in no particular order. However, in many real-world situations, the data we receive may be sorted in some way.

For instance, perhaps the rows of the data have been ordered by the date an account was created, or by some other logic. Therefore, it can be a good idea to shuffle the data before splitting. This way, any traits that might have been used for sorting can be expected to be consistent throughout the folds. Otherwise, the data in different folds may have different characteristics, possibly leading to different relationships between features and response. This can lead to a situation where model performance is uneven between the folds. In order to "mix up" the folds throughout all the row indices of a dataset, all we need to do is set the shuffle parameter to True:

k\_folds = StratifiedKFold(n\_splits=n\_folds, shuffle=True, random\_state=1)



Figure 4.19: Train/test splits for stratified k-folds with shuffling

With shuffling, the testing folds are spread out randomly, and fairly evenly, across the indices of the input data.

K-folds cross-validation is a widely used method in data science. However, the choice of how many folds to use depends on the particular dataset at hand. Using a smaller number of folds means that the amount of training data in each fold will be relatively small. Therefore, this increases the chances that the model will be underfit, as models generally work better when trained on more data. It's a good idea to try a few different numbers of folds and see how the mean and the variability of the k-folds testing score changes. Common numbers of folds can range anywhere from 4 or 5 to 10.

In the event of a very small dataset, it may be necessary to use as much data as possible for training in the cross-validation folds. In this scenario, you can use a method called leave-one-out cross-validation (LOOCV). In LOOCV, the testing set for each fold consists of a single sample. In other words, there will be as many folds as there are samples in the training data. For each iteration, the model is trained on all but one sample, and a prediction is made for that sample. The accuracy, or other performance metric, can then be constructed using these predictions.

Other concerns that relate to creation of a test set, such as choosing an out-of-time test set for problems where observations from the past must be used to predict future events, also apply to cross-validation.

In exercise 17, we saw that fitting a logistic regression on our training data led to overfitting. Indeed, the testing score (ROC AUC = 0.81) was substantially lower than the training score (ROC AUC = 0.94). We had essentially used very little or no regularization by setting the regularization parameter C to a relatively large value (1,000). Now we will see what happens when we vary C through a wide range of values.

Exercise 18: Reducing Overfitting on the Synthetic Data Classification Problem

This exercise is a continuation of Exercise 17, Generating and Modeling Synthetic Classification Data. Here, we will use a cross-validation procedure in order to find a good value for the hyperparameter C. We will do this by using only the training data, reserving the testing data for after model building is complete. Be prepared, this is a long exercise – but it will illustrate a general procedure that you will be able to use with many different kinds of machine learning models, so it is worth the time spent here. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found at ~~<http://bit.ly/2ZAy2Pr>~~.

1. Vary the value of the regularization parameter, C, to have it range from C = 1000 to C = 0.001. You can use the following snippets to do this.

First, define exponents, which will be powers of 10, as follows:

C\_val\_exponents = np.linspace(3,-3,13)

C\_val\_exponents

Here is the output of the preceding code:

array([ 3. , 2.5, 2. , 1.5, 1. , 0.5, 0. , -0.5, -1. , -1.5, -2. ,

-2.5, -3. ])

Now vary C by the powers of 10, as follows:

C\_vals = np.float(10)\*\*C\_val\_exponents

C\_vals

Here is the output of the preceding code:

array([1.00000000e+03, 3.16227766e+02, 1.00000000e+02, 3.16227766e+01,

1.00000000e+01, 3.16227766e+00, 1.00000000e+00, 3.16227766e-01,

1.00000000e-01, 3.16227766e-02, 1.00000000e-02, 3.16227766e-03,

1.00000000e-03])

It's generally a good idea to vary the regularization parameter by powers of 10, or by using a similar strategy, as training models can take substantial time, especially when using k-folds cross-validation. This gives you a good idea of how a wide range of C values impacts the bias-variance trade-off, without needing to train a very large number of models. In addition to the integer powers of 10, we also include points on the log10 scale that are about halfway between. If it seems like there is some interesting behavior in between these relatively widely spaced values, you can add more granular values for C in a smaller part of the range of possible values.

1. Import the roc\_curve class:

from sklearn.metrics import roc\_curve

We'll continue to use the ROC AUC score for assessing, training, and testing performance. Now that we have several values of C to try and several folds (in this case four) for the cross-validation, we will want to store the training and testing scores for each fold and for each value of C.

1. Define a function that takes the k\_folds cross-validation splitter, the array of C values (C\_vals), the model object (model), and the features and response variable (X and Y, respectively) as inputs, to explore different amounts of regularization with k-folds cross-validation. Use the following code:

def cross\_val\_C\_search(k\_folds, C\_vals, model, X, Y):

Note

The function we started in this step will return the ROC AUCs and ROC curve data. The return block will be written during a later step in the exercise. For now, you can simply write the preceding code as is, because we will be defining k\_folds, C\_vals, model, X, and Y as we progress in the exercise.

1. Within this function block, create a NumPy array to hold model performance data, with dimensions n\_folds by len(C\_vals):

n\_folds = k\_folds.n\_splits

cv\_train\_roc\_auc = np.empty((n\_folds, len(C\_vals)))

cv\_test\_roc\_auc = np.empty((n\_folds, len(C\_vals)))

Next, we'll store the arrays of true and false positive rates and thresholds that go along with each of these testing ROC AUC scores, in a list of lists.

Note

This is a convenient way to store all this information, as a list in Python can contain any kind of data, including another list. Here, each item in the list of lists will be a tuple holding the arrays of TPR, FPR, and the thresholds for each of the folds, for each of the C values. This should be more obvious when we access these arrays later in order to examine them.

1. Create a list of empty lists using [[]] and \*len(C\_vals) as follows:

cv\_test\_roc = [[]]\*len(C\_vals)

Using \*len(C\_vals) indicates that there should be a list of tuples of metrics (TPR, FPR, thresholds) for each value of C.

We have learned how to loop through the different folds for cross-validation in the preceding section. What we need to do now is write an outer loop, in which we will nest the cross-validation loop.

1. Create an outer loop for training and testing each of the k-folds for each value of C:

for c\_val\_counter in range(len(C\_vals)):

#Set the C value for the model object

model.C = C\_vals[c\_val\_counter]

#Count folds for each value of C

fold\_counter = 0

We can reuse the same model object that we have already, and simply set a new value of C within each run of the loop. Inside the loop of C values, we run the cross-validation loop. We begin by yielding the training and testing data row indices for each split.

1. Obtain the training and testing indices for each fold:

for train\_index, test\_index in k\_folds.split(X, Y):

1. Index the features and response variable to obtain the training and testing data for this fold using the following code:

X\_cv\_train, X\_cv\_test = X[train\_index], X[test\_index]

y\_cv\_train, y\_cv\_test = Y[train\_index], Y[test\_index]

The training data for the current fold is then used to train the model.

1. Fit the model on the training data, as follows:

model.fit(X\_cv\_train, y\_cv\_train)

This will effectively "reset" the model from whatever the previous coefficients and intercept were, to reflect the training on this new data.

The training and testing ROC AUC scores are then obtained, as well as the arrays of TPRs, FPRs, and thresholds that go along with the testing data.

1. Obtain the training ROC AUC score:

y\_cv\_train\_predict\_proba = model.predict\_proba(X\_cv\_train)

cv\_train\_roc\_auc[fold\_counter, c\_val\_counter] = \

roc\_auc\_score(y\_cv\_train, y\_cv\_train\_predict\_proba[:,1])

1. Obtain the testing ROC AUC score:

y\_cv\_test\_predict\_proba = model.predict\_proba(X\_cv\_test)

cv\_test\_roc\_auc[fold\_counter, c\_val\_counter] = \

roc\_auc\_score(y\_cv\_test, y\_cv\_test\_predict\_proba[:,1])

1. Obtain the testing ROC curves for each fold using the following code:

this\_fold\_roc = roc\_curve(y\_cv\_test, y\_cv\_test\_predict\_proba[:,1])

cv\_test\_roc[c\_val\_counter].append(this\_fold\_roc)

We will use a fold counter to keep track of the folds that are incremented, and once outside the cross-validation loop, we print a status update to standard output. Whenever performing long computational procedures, it's a good idea to periodically print the status of the job, so that you can monitor its progress and confirm that things are still working correctly. This cross-validation procedure will likely take only a few seconds on your laptop, but for longer jobs this can be especially reassuring.

1. Increment the fold counter using the following code:

fold\_counter += 1

1. Write the following code to indicate the progress of execution for each value of C:

print('Done with C = {}'.format(lr\_syn.C))

1. Write the code to return the ROC AUCs and ROC curve data and finish the function:

return cv\_train\_roc\_auc, cv\_test\_roc\_auc, cv\_test\_roc

Note that we will continue to use the split into four folds that we illustrated previously, but you are encouraged to try this procedure with different numbers of folds to compare the effect.

We have covered a lot of material in the preceding steps. You may want to take a few moments to review this with your classmates in order to make sure that you understand each part. Running the function is comparatively simple. That is the beauty of a well-designed function – all the complicated parts get abstracted away, allowing you to concentrate on usage.

1. Run the function we’ve designed to examine cross-validation performance, with the C values that we previously defined, and by using the model and data we were working with in the previous exercise. Use the following code:

cv\_train\_roc\_auc, cv\_test\_roc\_auc, cv\_test\_roc = \

cross\_val\_C\_search(n\_folds, C\_vals, lr\_syn, X\_syn\_train, y\_syn\_train)

When you run this code, you should see the following output populate below the code cell as the cross-validation is completed for each value of C.

Done with C = 1000.0

Done with C = 316.22776601683796

Done with C = 100.0

Done with C = 31.622776601683793

Done with C = 10.0

Done with C = 3.1622776601683795

Done with C = 1.0

Done with C = 0.31622776601683794

Done with C = 0.1

Done with C = 0.03162277660168379

Done with C = 0.01

Done with C = 0.0031622776601683794

Done with C = 0.001

So, what do the results of the cross-validation look like? There are a few ways to examine this. It is useful to look at the performance of each fold individually, so that you can see how variable the results are.

This tells you how different subsets of your data perform as test sets, leading to a general idea of the range of performance you can expect from the unseen test set. What we're interested in here is whether or not we are able to use regularization to alleviate the overfitting that we saw. We know that using C = 1,000 led to overfitting – we know this from comparing the training and testing scores. But what about the other C values that we've tried? A good way to visualize this will be to plot the training and testing scores on the y-axis and the values of C on the x-axis.

1. Loop over each of the folds to view their results individually by using the following code:

for this\_fold in range(n\_folds):

plt.plot(C\_val\_exponents, cv\_train\_roc\_auc[this\_fold], '-o',

color=cmap(this\_fold),

label='Training fold {}'.format(this\_fold+1))

plt.plot(C\_val\_exponents, cv\_test\_roc\_auc[this\_fold], '-x',

color=cmap(this\_fold),

label='Testing fold {}'.format(this\_fold+1))

plt.ylabel('ROC AUC')

plt.xlabel('log$\_{10}$(C)')

plt.legend(loc = [1.1, 0.2])

plt.title('Cross validation scores for each fold')

You will obtain the following output:

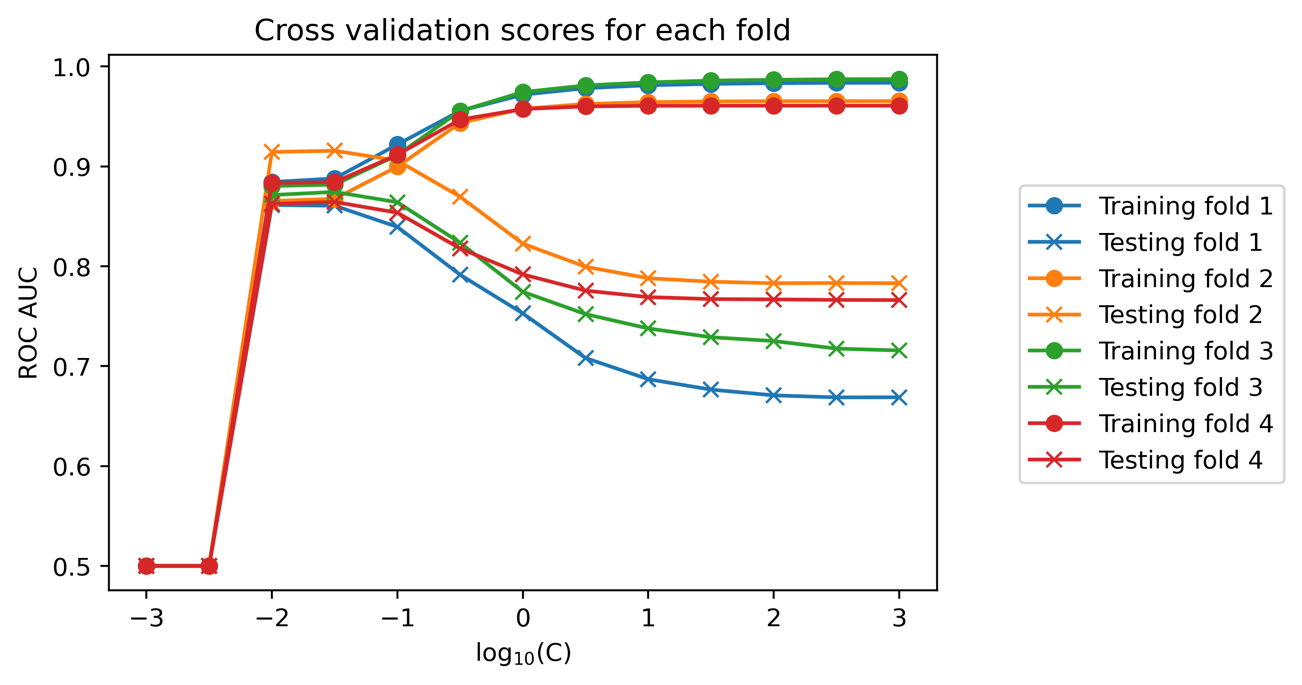


Figure 4.20: The training and testing scores for each fold and C-value

We can see that for each fold of the cross-validation, as C decreases, the training performance also decreases. However, at the same time, the testing performance increases. For some folds and values of C, the testing ROC AUC score actually exceeds that of the training data, while for others, these two metrics simply come closer together. In all cases, we can say that the C values of 10-1.5 and 10-2 appear to have a similar testing performance, which is substantially higher than the testing performance of C = 103. So, it appears that regularization has successfully addressed our overfitting problem.

But what about the lower values of C? For values that are lower than 10-2, the ROC AUC metric suddenly drops to 0.5. As you know, this value means that the classification model is essentially useless, performing no better than a coin flip. You are encouraged to check on this later when exploring how regularization affects the coefficient values; however, this is what happens when so much L1 regularization is applied that all model coefficients shrink to 0. Obviously, such models are not useful to us, as they encode no information about the relationship between the features and response variable.

Looking at the training and testing performance of each k-folds split is helpful in gaining insight into the variability of model performance that may be expected when the model is scored on new, unseen data. But in order to summarize the results of the k-folds procedure, a common approach is to average the performance metric over the folds, for each value of the hyperparameter being considered. We'll perform this in the next step.

1. Plot the mean of training and testing ROC AUC scores for each C value using the following code:

plt.plot(C\_val\_exponents, np.mean(cv\_train\_roc\_auc, axis=0), '-o',

label='Average training score')

plt.plot(C\_val\_exponents, np.mean(cv\_test\_roc\_auc, axis=0), '-x',

label='Average testing score')

plt.ylabel('ROC AUC')

plt.xlabel('log$\_{10}$(C)')

plt.legend()

plt.title('Cross validation scores averaged over all folds')

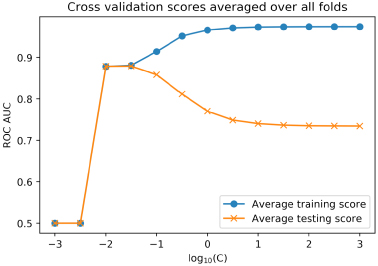


Figure 4.21: The average training and testing scores across cross-validation folds

From this plot, it's clear that C = 10-1.5 and 10-2 are the best values of C. There is little or no overfitting here, as the average training and testing scores are nearly the same. You could search a finer grid of C values (that is C = 10-1.1, 10-1.2, and so on) in order to more precisely locate a C value. However, from our graph we can see that either C = 10-1.5 or C = 10-2 will likely be good solutions. We will move forward with C = 10-1.5.

Examining the summary metric of ROC AUC is a good way to get a quick idea of how models will perform. However, for any real-world business application, you will often need to choose a specific threshold, which goes along with specific true and false positive rates. These will be needed to use the classifier to make the required "yes" or "no" decision, which in our case study is a prediction of whether or not an account will default. For this reason, it is useful to look at the ROC curves across the different folds of the cross-validation. To facilitate this, the preceding function has been designed to return the true and false positive rates, and thresholds, for each testing fold and value of C, in the cv\_test\_roc list of lists. First, we need to find the index of the outer list that corresponds to the C value that we've chosen, 10-1.5.

To accomplish this, we could simply look at our list of C values and count by hand, but it's safer to do this programmatically by finding the index of the non-zero element of a Boolean array as is shown in the next step.

1. Use a Boolean array to find the index where C = 10-1.5 and convert to an integer data type with this code:

best\_C\_val\_bool = C\_val\_exponents == -1.5  
best\_C\_val\_bool.astype(int)

Here is the output of the preceding code:

array([0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0])

1. Convert the integer version of the Boolean array into a single integer index using the nonzero function with this code:

best\_C\_val\_ix = np.nonzero(best\_C\_val\_bool.astype(int)) best\_C\_val\_ix[0][0]

best\_C\_val\_ix[0][0]

Here is the output of the preceding code:

9

We have now successfully located the C value that we wish to use.

1. Access the true and false positive rates in order to plot the ROC curves for each fold:

for this\_fold in range(n\_folds):

fpr = cv\_test\_roc[best\_C\_val\_ix[0][0]][this\_fold][0]

tpr = cv\_test\_roc[best\_C\_val\_ix[0][0]][this\_fold][1]

plt.plot(fpr, tpr, label='Fold {}'.format(this\_fold+1))

plt.xlabel('False positive rate')

plt.ylabel('True positive rate')

plt.title('ROC curves for each fold at C = $10^{-1.5}$')

plt.legend()

You will obtain the following output:

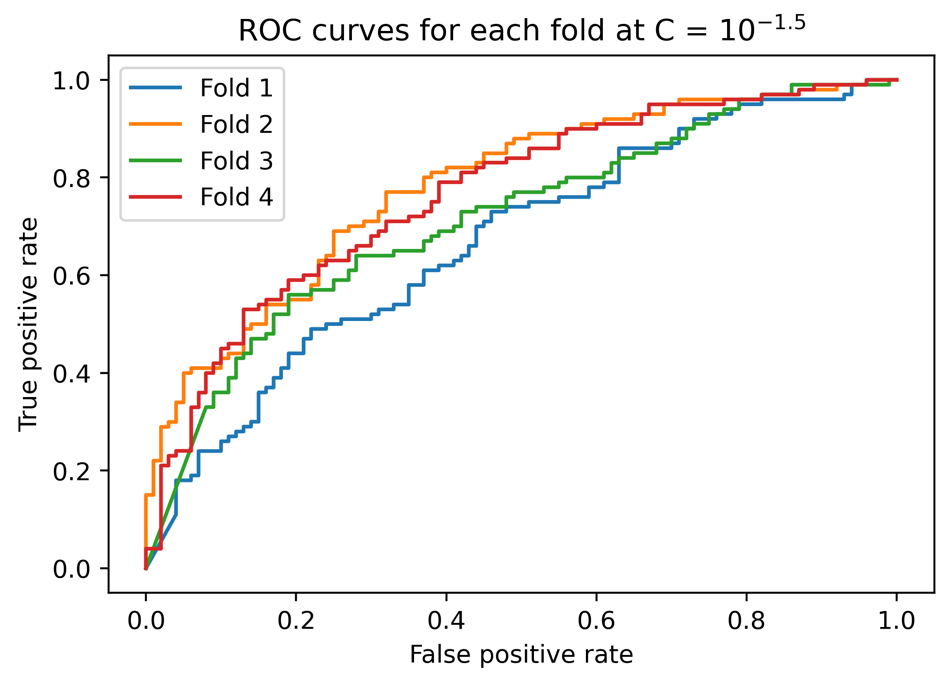


Figure 4.22: ROC curves for each fold

It appears that there is a fair amount of variability in the ROC curves. For example, if for some reason we want to limit the false positive rate to 40%, then from the plot it appears that we may be able to achieve a true positive rate of anywhere from approximately 60% to 80%. You can find the exact values by examining the arrays that we have plotted. This gives you an idea of how much variability in performance can be expected when deploying the model on new data. Generally, the more training data that is available, then the less variability there will be between the folds of cross-validation, so this could also be a sign that it would be a good idea to collect additional data, especially if the variability between training folds seems unacceptably high. You also may wish to try different numbers of folds with this procedure, to see the effect on variability of results between folds.

While normally we would try other models on our synthetic data problem, such as a random forest or support vector machine, if we imagine that in cross-validation, logistic regression proved to be the best model, we would decide to make it our final choice. When the final model is selected, all the training data can be used to fit the model, using the hyperparameters chosen with cross-validation. It's best to use as much data as possible in model fitting, as models typically work better when trained on more data.

1. Train the logistic regression on all the training data from our synthetic problem and compare the training and testing scores, using the held-out test set as shown in the following steps.

Note

This is the last step in the model selection process. You should only use the unseen test set after your choice of model and hyperparameters are considered finished, otherwise it will not be "unseen."

1. Set the C value and train the model on all the training data with this code:

lr\_syn.C = 10\*\*(-1.5)

lr\_syn.fit(X\_syn\_train, y\_syn\_train)

Here is the output of the preceding code:

LogisticRegression(C=0.03162277660168379, penalty='l1', random\_state=1, solver='liblinear'))

1. Obtain predicted probabilities and the ROC AUC score for the training data with this code:

y\_syn\_train\_predict\_proba = lr\_syn.predict\_proba(X\_syn\_train)

roc\_auc\_score(y\_syn\_train, y\_syn\_train\_predict\_proba[:,1])

Here is the output of the preceding code

0.8802812499999999

1. Obtain predicted probabilities and the ROC AUC score for the testing data with this code:

y\_syn\_test\_predict\_proba = lr\_syn.predict\_proba(X\_syn\_test)

roc\_auc\_score(y\_syn\_test, y\_syn\_test\_predict\_proba[:,1])

Here is the output of the preceding code

0.8847884788478848

Here, we can see that by using regularization, the model training and testing scores are similar, indicating that the overfitting problem has been greatly reduced. The training score is lower, since we have introduced bias into the model at the expense of variance. However, this is okay, since the testing score, which is the most important part, is higher. The out-of-sample testing score is what matters for predictive capability. You are encouraged to check that these training and testing scores are similar to those from the cross-validation procedure by printing the values from the arrays that we plotted previously; you should find that they are.

Note

In a real-world project, before delivering this model to your client for production use, you may wish to train the model on all the data that you were given, including the unseen test set. This follows the idea that the more data a model has seen, the better it is likely to perform in practice. However, some practitioners prefer to only use models that have been tested, meaning you would deliver the model trained only on the training data, not including the test set.

We know that L1 regularization works by decreasing the magnitude (that is, absolute value) of coefficients of the logistic regression. It can also set some coefficients to zero, therefore performing feature selection. In the next step, we will determine how many coefficients were set to zero.

1. Access the coefficients of the trained model and determine how many do not equal zero (!= 0) with this code:

sum((lr\_syn.coef\_ != 0)[0])

The output should be:

2

This code takes the sum of a Boolean array indicating the locations of non-zero coefficients, so it shows how many coefficients in the model did not get set to zero by L1 regularization. Only 2 of the 200 features were selected!

1. Examine the value of the intercept using this code:

lr\_syn.intercept\_

The output should be:

array([0.])

This shows that the intercept was regularized to 0.

In this exercise, we accomplished several goals. We used the k-folds cross-validation procedure to tune the regularization hyperparameter. We saw the power of regularization for reducing overfitting, and in the case of L1 regularization in logistic regression, selecting features.

Many machine learning algorithms offer some type of feature selection capability. Many also require the tuning of hyperparameters. The function here that loops over hyperparameters, and performs cross-validation, is a powerful concept that generalizes to other models. Scikit-learn offers functionality to make this process easier; in particular, the sklearn.model\_selection.GridSearchCV procedure, which applies cross-validation to a grid search over hyperparameters. A grid search can be helpful when there are multiple hyperparameters to tune, by looking at all combinations of the ranges of different hyperparameters that you specify. A randomized grid search can speed up this process by randomly choosing a smaller number of combinations when an exhaustive grid search would take too long. Once you are comfortable with the concepts illustrated here, you are encouraged to streamline your workflow with convenient functions like these.

Options for Logistic Regression in Scikit-Learn

We have used and discussed most of the options that you may supply to scikit-learn when instantiating or tuning the hyperparameters of a LogisticRegression model class. Here, we list them all and give some general advice on their usage:

| Parameter | Possible values | Notes and advice for choosing |
| --- | --- | --- |
| penalty | string, ‘l1’ , ‘l2’, ‘elasticnet’, or ‘none’ | L1 (lasso) or L2 (ridge) regularization of coefficients. L1 performs feature selection, while L2 does not. Elastic-net is a blend of L1 and L2. The best overall model performance should be assessed by trying all options. |
| dual | bool, True or False | This has to do with the optimization algorithm used to find coefficients. The documentation says “only implemented for l2 penalty with liblinear solver. Prefer dual=False when n\_samples > n\_features.” |
| tol | float (decimal number) | Determines the size of the change in values for the optimization algorithm to stop. This is one way to control how long the optimization runs for, and how close to the ideal value the solution is. |
| C | float | The regularization parameter for L1 or L2 penalties. Smaller values mean more regularization. This needs to be determined using a validation set, or cross-validation. |
| fit\_intercept | bool | Whether or not an intercept term should be estimated. Unless you are sure you don’t need an intercept, it’s probably best to have one. |
| intercept\_scaling | float | Can be used to avoid regularizing the intercept, an undesirable practice, when using the liblinear solver. |
| class\_weight | dictionary specify weight for each class, string ‘balanced’, or None | Whether or not to weight different classes during the model training process. Otherwise all samples will be considered “equally important” when fitting the model. Can be useful for imbalanced data sets: try using ‘balanced’ in this case. |
| random\_state | int | Seed for random number generator used by certain solver algorithms. |
| solver | string (‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’, ‘saga’) | Select the type of optimization algorithm used to estimate the model parameters. See earlier discussion in this chapter or the documentation for the relative strengths and weaknesses of different solvers. |
| max\_iter | int | The maximum number of iterations for the solution algorithm, which controls how close to the ideal parameters the solution is. If you get a warning that the solution algorithm did not converge, you can try increasing this. |
| multi\_class | string (‘ovr’, ‘multinomial’, ‘auto’) | Various strategies for multiclass classification, beyond the scope of this book. |
| verbose | int | Controls the nature of the output to the terminal, during the optimization procedure. |
| warm\_start | bool | If re-using the same model object for multiple trainings, whether or not to use the previous solution as the starting point for the next optimization procedure. |
| n\_jobs | int or None | Number of processers to use for parallel processing, in the case of ‘ovr’ multiclass classification. |
| l1\_ratio | float | Parameter controlling the relative contributions of L1 and L2 regularization when using the elastic-net penalty. |

Figure 4.23: A complete list of options for the logistic regression model in scikit-learn

If you are in doubt regarding which option to use for logistic regression, we recommend you consult the scikit-learn documentation for further guidance (<https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression>). Some options, such as the regularization parameter C, or the choice of a penalty for regularization, will need to be explored through the cross-validation process. Here, as with many choices to be made in data science, there is no universal approach that will apply to all datasets. The best way to see which options to use with a given dataset is to try several of them and see which gives the best out-of-sample performance. Cross-validation offers you a robust way to do this.

Scaling Data, Pipelines, and Interaction Features in Scikit-Learn

Scaling Data

Compared to the synthetic data we were just working with, the case study data is relatively large. If we want to use L1 regularization, then according to the official documentation (<https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression>), we ought to use the saga solver. However, this solver is not robust to unscaled datasets. So, we need to be sure to scale the data. This is also a good idea whenever doing regularization, so all the features are on the same scale and are equally penalized by the regularization process. A simple way to make sure that all the features have the same scale is to put them all through the transformation of subtracting the minimum, and dividing by the range from minimum to maximum. This transforms each feature so that it will have a minimum of 0 and a maximum of 1. To instantiate the MinMaxScaler scaler which does this, we can use the following code:

from sklearn.preprocessing import MinMaxScaler

min\_max\_sc = MinMaxScaler()

Pipelines

Previously, we used a logistic regression model in the cross-validation loop. However, now that we're scaling data, what new considerations are there? The scaling is effectively "learned" from the minimum and maximum values of the training data. After this, a logistic regression model would be trained on data scaled by the extremes of the model training data. However, we won't know the minimum and maximum values of the new, unseen data. So, following the philosophy of making cross-validation an effective indicator of model performance on unseen data, we need to use the minimum and maximum values of the training data in each cross-validation fold in order to scale the testing data in that fold, before making predictions on the testing data. Scikit-learn has functionality to facilitate the combination of several training and testing steps for situations like this: the Pipeline. Our pipeline will consist of two steps: the scaler and the logistic regression model. These can both be fit on the training data, and then be used to make predictions on the testing data. The process of fitting a pipeline is executed as a single step in the code, so all the parts of the pipeline are fit at once in this sense. Here is how a Pipeline is instantiated:

from sklearn.pipeline import Pipeline

scale\_lr\_pipeline = Pipeline(steps=[('scaler', min\_max\_sc), ('model', lr)])

Interaction Features

Considering the case study data, do you think a logistic regression model with all possible features would be overfit or underfit? You can think about this from the perspective of rules of thumb such as the "rule of 10", and the number of features (17) versus samples (26,664) that we have. Alternatively, you can consider all the work we've done so far with these data. For instance, we've had a chance to visualize all the features and ensure they make sense. Since there are relatively few features, and we have relatively high confidence that they are high quality because of our data exploration work, we are in a different situation than with the synthetic data exercises in this chapter, where we had a large number of features about which we knew relatively little. So, it may be that overfitting will be less of an issue with our case study at this point, and the benefits of regularization may not be significant.

In fact, it may be that we will underfit the model using only the 17 features that came with the data. One strategy to deal with this is to engineer new features. Some simple feature engineering techniques we've discussed include interaction and polynomial features. Polynomials may not make sense given the way that some of the data have been encoded; for example, -12 = 1, which may not be sensible for PAY\_1. However, we may wish to try creating interaction features to capture the relationships between features. PolynomialFeatures can be used to create interaction features only, without polynomial features. Example code is as follows:

make\_interactions = PolynomialFeatures(degree=2, interaction\_only=True, include\_bias=False)

Here, degree represents the degree of the polynomial features, interaction\_only takes a Boolean value (setting it to True indicates that only interaction features will be created), and so does include\_bias, which adds an intercept to the model (the default value is False, which is correct here as the logistic regression model will add an intercept).

Activity 4: Cross-Validation and Feature Engineering with the Case Study Data

In this activity, we'll apply the knowledge of cross-validation and regularization that we've learned in this chapter to the case study data. We'll perform basic feature engineering. In order to estimate parameters for the regularized logistic regression model for the case study data, which is larger in size than the synthetic data that we've worked with, we'll use the saga solver. In order to use this solver, and for the purpose of regularization, we'll need to scale our data as part of the modeling process, leading us to the use of Pipelines in scikit-learn. Once you have completed the activity, you should obtain improved cross-validation testing performance with the use of interaction features, as shown in the following figure:

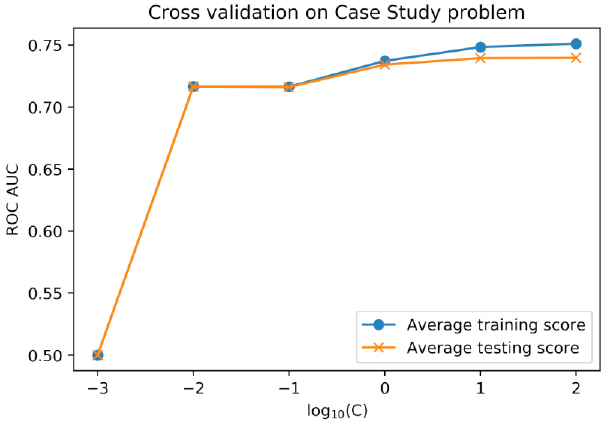


Figure 4.24: Improved model testing performance

Perform the following steps to complete the activity:

Note

The code and the resulting output for this activity have been loaded in a Jupyter Notebook and can be found at ~~<http://bit.ly/2Z53aX4>~~.

1. Select the features from the DataFrame of the case study data.

You can use the list of feature names that we've already created in this chapter. But be sure not to include the response variable, which would be a very good (but entirely inappropriate) feature!

1. Make a train/test split using a random seed of 24.

We'll use this going forward and reserve this testing data as the unseen test set. By specifying the random seed, we can easily create separate notebooks with other modeling approaches, using the same training data.

1. Instantiate the MinMaxScaler to scale the data.
2. Instantiate a logistic regression model with the saga solver, L1 penalty, and set max\_iter to 1,000 as we want the solver to have enough iterations to find a good solution.
3. Import the Pipeline class and create a Pipeline with the scaler and the logistic regression model, using the names 'scaler' and 'model' for the steps, respectively.
4. Use the get\_params and set\_params methods to see how to view the parameters from each stage of the pipeline and change them.
5. Create a smaller range of C values to test with cross-validation, as these models will take longer to train and test with more data than our previous exercise; we recommend C = [102, 10, 1, 10-1, 10-2, 10-3].
6. Make a new version of the cross\_val\_C\_search function, called cross\_val\_C\_search\_pipe. Instead of the model argument, this function will take a pipeline argument. The changes inside the function will be to set the C value using set\_params(model\_\_C = <value you want to test>) on the pipeline, replacing the model with the pipeline for the fit and predict\_proba methods, and accessing the C value using pipeline.get\_params()['model\_\_C'] for the printed status update.
7. Run this function as in the previous exercise, but using the new range of C values, the pipeline you created, and the features and response variable from the training split of the case study data.

You may see warnings here, or in later steps, about the non-convergence of the solver; you could experiment with the tol or max\_iter options to try and achieve convergence, although the results you obtain with max\_iter = 1000 are likely to be sufficient.

1. Plot the average training and testing ROC AUC across folds, for each C value.
2. Create interaction features for the case study data and confirm that the number of new features makes sense.
3. Repeat the cross-validation procedure and observe the model performance now.

Note that this will take substantially more time, due to the larger number of features, but it will probably take less than 10 minutes. So, does the average cross-validation testing performance improve with the interaction features? Is regularization useful?

Note

The solution for this activity can be found on page 339.

Summary

In this chapter, we introduced the final details of logistic regression and continued to use scikit-learn to fit logistic regression models. We gained more visibility into how the model fitting process works by learning about the concept of a cost function, which is minimized by the gradient descent procedure to estimate model parameters during model fitting.

We also learned of the need for regularization, by introducing the concepts of underfitting and overfitting. In order to reduce overfitting, we saw how to adjust the cost function to regularize the coefficients of a logistic regression model using an L1 or L2 penalty. We used cross-validation to select the amount of regularization, by tuning the regularization hyperparameter. To reduce underfitting, we saw how to do some simple feature engineering, with interaction features for the case study data.

We are now familiar with some of the most important concepts in machine learning. We have, so far, only used a very basic classification model: logistic regression. However, as you increase your toolbox of models that you know how to use, you will find that the concepts of overfitting, underfitting, the bias-variance trade-off, and hyperparameter tuning will come up again and again. These ideas, as well as convenient scikit-learn implementations of the cross-validation functions that we wrote in this chapter, will help us through our exploration of more advanced prediction methods.

In the next chapter, we will learn about decision trees, an entirely different type of predictive model than logistic regression, and the random forests that are based on them. However, we will use the same concepts that we learned here, cross-validation and hyperparameter search, to tune these models.

Chapter 4: The Bias-Variance Trade-off

Activity 4: Cross-Validation and Feature Engineering with the Case Study Data

1. Select out the features from the DataFrame of the case study data.

You can use the list of feature names that we've already created in this chapter. But be sure not to include the response variable, which would be a very good (but entirely inappropriate) feature:

features = features\_response[:-1]

X = df[features].values

1. Make a train/test split using a random seed of 24:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, df['default payment next month'].values,

test\_size=0.2, random\_state=24)

We'll use this going forward and reserve this testing data as the unseen test set. By specifying the random seed, we can easily create separate notebooks with other modeling approaches, using the same training data.

1. Instantiate the MinMaxScaler to scale the data, as shown in the following code:

from sklearn.preprocessing import MinMaxScaler

min\_max\_sc = MinMaxScaler()

1. Instantiate a logistic regression model with the saga solver, L1 penalty, and set max\_iter to 1,000 as we'd like to allow the solver enough iterations to find a good solution:

lr = LogisticRegression(solver='saga', penalty='l1', max\_iter=1000)

1. Import the Pipeline class and create a Pipeline with the scaler and the logistic regression model, using the names 'scaler' and 'model' for the steps, respectively:

from sklearn.pipeline import Pipeline

scale\_lr\_pipeline = Pipeline(

steps=[('scaler', min\_max\_sc), ('model', lr)])

1. Use the get\_params and set\_params methods to see how to view the parameters from each stage of the pipeline and change them (execute each of the following lines in a separate cell in your notebook and observe the output):

scale\_lr\_pipeline.get\_params()

scale\_lr\_pipeline.get\_params()['model\_\_C']

scale\_lr\_pipeline.set\_params(model\_\_C = 2)

1. Create a smaller range of C values to test with cross-validation, as these models will take longer to train and test with more data than our previous exercise; we recommend C = [102, 10, 1, 10-1, 10-2, 10-3]:

C\_val\_exponents = np.linspace(2,-3,6)

C\_vals = np.float(10)\*\*C\_val\_exponents

1. Make a new version of the cross\_val\_C\_search function, called cross\_val\_C\_search\_pipe. Instead of the model argument, this function will take a pipeline argument. The changes inside the function will be to set the C value using set\_params(model\_\_C = <value you want to test>) on the pipeline, replacing model with the pipeline for the fit and predict\_proba methods, and accessing the C value using pipeline.get\_params()['model\_\_C'] for the printed status update.

The changes are as follows:

def cross\_val\_C\_search\_pipe(k\_folds, C\_vals, pipeline, X, Y):

##[…]

pipeline.set\_params(model\_\_C = C\_vals[c\_val\_counter])

##[…]

pipeline.fit(X\_cv\_train, y\_cv\_train)

##[…]

y\_cv\_train\_predict\_proba = pipeline.predict\_proba(X\_cv\_train)

##[…]

y\_cv\_test\_predict\_proba = pipeline.predict\_proba(X\_cv\_test)

##[…]

print('Done with C = {}'.format(pipeline.get\_params()['model\_\_C']))

Note

For the complete code, refer to ~~<http://bit.ly/2ZAy2Pr>~~.

1. Run this function as in the previous exercise, but using the new range of C values, the pipeline you created, and the features and response variable from the training split of the case study data. You may see warnings here, or in later steps, about the non-convergence of the solver; you could experiment with the tol or max\_iter options to try and achieve convergence, although the results you obtain with max\_iter = 1000 are likely to be sufficient. Here is the code to do this:

cv\_train\_roc\_auc, cv\_test\_roc\_auc, cv\_test\_roc = \

cross\_val\_C\_search\_pipe(k\_folds, C\_vals, scale\_lr\_pipeline,

X\_train, y\_train)

You will obtain the following output:

Done with C = 100.0

Done with C = 10.0

Done with C = 1.0

Done with C = 0.1

Done with C = 0.01

Done with C = 0.001

1. Plot the average training and testing ROC AUC across folds, for each C value, using the following code:

plt.plot(C\_val\_exponents, np.mean(cv\_train\_roc\_auc, axis=0), '-o',

label='Average training score')

plt.plot(C\_val\_exponents, np.mean(cv\_test\_roc\_auc, axis=0), '-x',

label='Average testing score')

plt.ylabel('ROC AUC')

plt.xlabel('log$\_{10}$(C)')

plt.legend()

plt.title('Cross validation on Case Study problem')

np.mean(cv\_test\_roc\_auc, axis=0)

You will obtain the following output:

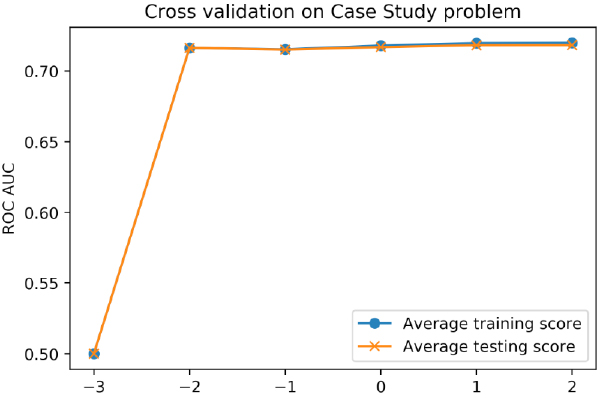


Figure 6.54: Cross-validation testing performance

You should notice that regularization does not impart much benefit here, as may be expected: for lower C values, which correspond to stronger regularization, model testing (as well as training) performance decreases. While we are able to increase model performance over our previous efforts by using all the features available, it appears there is no overfitting going on. Instead, the training and testing scores are about the same. Instead of overfitting, it's possible that we may be underfitting. Let's try engineering some interaction features to see if they can improve performance.

1. Create interaction features for the case study data and confirm that the number of new features makes sense using the following code:

from sklearn.preprocessing import PolynomialFeatures

make\_interactions = PolynomialFeatures(degree=2, interaction\_only=True, include\_bias=False)

X\_interact = make\_interactions.fit\_transform(X)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X\_interact, df['default payment next month'].values,

test\_size=0.2, random\_state=24)

print(X\_train.shape)

print(X\_test.shape)

You will obtain the following output:

(21331, 153)

(5333, 153)

From this you should see the new number of features is 153, which is 17 + "17 choose 2" = 17 + 136 = 153. The "17 choose 2" part comes from choosing all possible combinations of 2 features to interact from the 17 original features.

1. Repeat the cross-validation procedure and observe the model performance now; that is, repeat Steps 9 and 10. Note that this will take substantially more time, due to the larger number of features, but it will probably take less than 10 minutes.

You will obtain the following output:

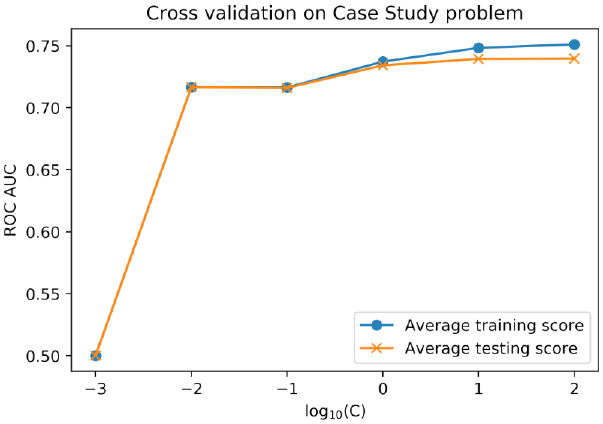


Figure 6.55: Improved cross-validation testing performance from adding interaction features

So, does the average cross-validation testing performance improve with the interaction features? Is regularization useful?

Engineering the interaction features increases the best model testing score to about ROC AUC = 0.74 on average across the folds, from about 0.72 without including interactions. These scores happen at C = 100, that is, with negligible regularization. On the plot of training versus testing scores for the model with interactions, you can see that the training score is a bit higher than the testing score, so it could be said that some amount of overfitting is going on. However, we cannot increase the testing score through regularization here, so this may not be a problematic instance of overfitting. In most cases, whatever strategy yields the highest testing score is the best strategy. In summary, adding interaction features improved cross-validation performance, and regularization appears not to be useful for the case study at this point, using a logistic regression model.

We will reserve the step of fitting on all the training data for later, when we've tried other models in cross-validation to find the best model.