3

Overview

We’re ready to start learning how to evaluate features quickly and efficiently, in order to know which ones will probably be most important for a machine learning model. Once we get a taste for this, we’ll explore the inner workings of logistic regression so you can continue your journey to mastery of this fundamental technique. After reading this chapter, you should know how to make a correlation plot of many features and a response variable, and how to interpret logistic regression as a linear model.

Introduction

In the previous chapter, we developed a few example machine learning models using scikit-learn, to get familiar with how it works. However, the features we used, EDUCATION and LIMIT\_BAL, were not chosen in a systematic way.

In this chapter, we will start to develop techniques that can be used to assess features for their usefulness in modeling. This will enable you to make a quick pass over all candidate features, to have an idea of which will be the most important. For the most promising features, we will see how to create visual summaries that serve as useful communication tools.

Next, we will begin our detailed examination of logistic regression. We'll learn why logistic regression is considered to be a linear model, even if the formulation involves some non-linear functions. We’ll learn what a decision boundary is and see that as a key consequence of its linearity, the decision boundary of logistic regression could make it difficult to accurately classify the response variable. Along the way, we'll get more familiar with Python, using list comprehensions and writing functions.

Examining the Relationships between Features and the Response Variable

In order to make accurate predictions of the response variable, good features are necessary. We need features that are clearly linked to the response variable in some way. Thus far, we've examined the relationship between a couple of features and the response variable, either by calculating the groupby/mean of a feature and the response variable, or using individual features in a model and examining performance. However, we have not yet done a systematic exploration of how all the features relate to the response variable. We will do that now and begin to capitalize on all the hard work we put in when we were exploring the features and making sure the data quality was good.

A popular way of getting a quick look at how all the features relate to the response variable, as well as how the features are related to each other, is by using a correlation plot. We will first create a correlation plot for the case study data, then discuss how to interpret it, along with some mathematical details.

In order to create a correlation plot, the necessary inputs include all features that we plan to explore, as well as the response variable. Since we are going to use most of the column names from the DataFrame for this, a quick way to get the appropriate list in Python is to start with all the column names and remove those that we don't want. As a preliminary step, we start a new notebook for this chapter and load packages and the cleaned data from Chapter 1, Data Exploration and Cleaning, with this code:

import numpy as np #numerical computation

import pandas as pd #data wrangling

import matplotlib.pyplot as plt #plotting package

#Next line helps with rendering plots

%matplotlib inline

import matplotlib as mpl #add'l plotting functionality

import seaborn as sns #a fancy plotting package

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

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

Note

The path to your file of cleaned data may be different, depending on where you saved it in Chapter 1, Data Exploration and Cleaning.

Notice that this notebook starts out in a very similar way to the previous chapter's notebook, except we also import the Seaborn package, which has many convenient plotting features that build on Matplotlib. Now let's make a list of all the columns of the DataFrame and look at the first and last five:

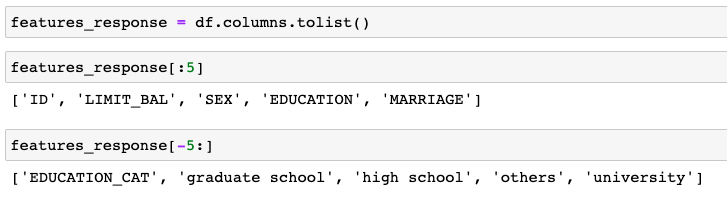


Figure 3.1: Get a list of column names

Recall that we are not to use the gender variable due to ethical concerns, and we learned that PAY\_2, PAY\_3,…, PAY\_6 are incorrect and should be ignored. Also, we are not going to examine the one-hot encoding we created from the EDUCATION variable, since the information from those columns is already included in the original feature, at least in some form. We will just use the EDUCATION feature directly. Finally, it makes no sense to use ID as a feature, since this is simply a unique account identifier and has nothing to do with the response variable. Let's make another list of column names that are neither features nor the response. We want to exclude these from our analysis:

items\_to\_remove = ['ID', 'SEX',\

                   'PAY\_2', 'PAY\_3', 'PAY\_4', 'PAY\_5', 'PAY\_6',\

                   'EDUCATION\_CAT',\

                   'graduate school', 'high school', 'none',\

                   'others', 'university']

To have a list of column names that consists only of the features and response we will use, we want to remove the names in items\_to\_remove from the current list contained in features\_response. There are several ways to do this in Python. We will use this opportunity to learn about a particular way of building a list in Python, called a list comprehension. When people talk about certain constructions as being Pythonic, or idiomatic to the Python language, list comprehensions are often one of the things that are mentioned.

What is a list comprehension? Conceptually, it is basically the same as a for loop. However, list comprehensions enable the creation of lists, which may be spread across several lines in an actual for loop, to be written in one line. They are also slightly faster than for loops, due to optimizations within Python. While this likely won't save us much time here, this is a good chance to become familiar with them. Here is an example list comprehension:

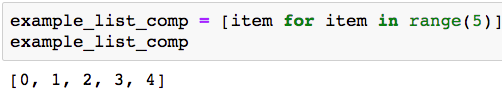


Figure 3.2: Example of a list comprehension

That's all there is to it!

We can also use additional clauses to make the list comprehensions flexible. For example, we can use them to reassign the features\_response variable with a list containing everything that's not in the list of strings we wish to remove:

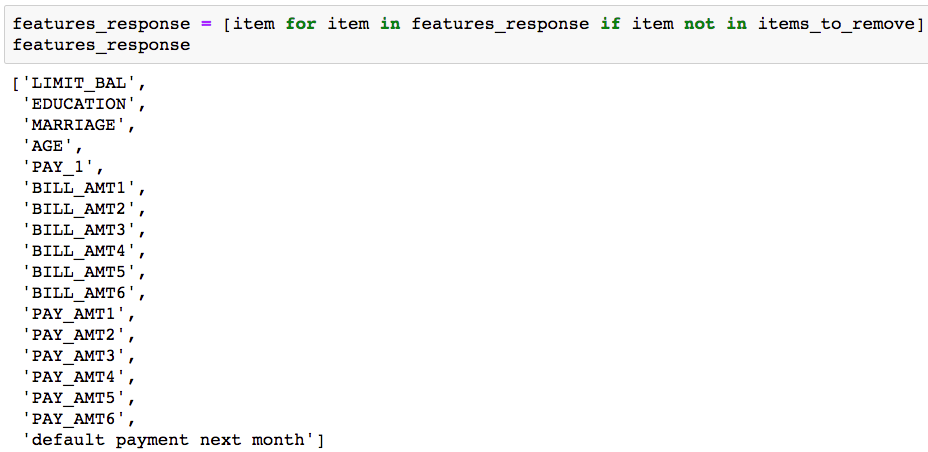


Figure 3.3: Using a list comprehension to prune down the column names

The use of if and not in within the list comprehension is fairly self-explanatory. Easy readability in structures such as list comprehensions is one of the reasons for the popularity of Python.

Note

The Python documentation (<https://docs.python.org/3/tutorial/datastructures.html>) defines list comprehensions as the following:

"A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses."

Thus, list comprehensions can enable you to do things with less code, in a way that is usually pretty readable and understandable.

Pearson Correlation

Now we are ready to create our correlation plot. Underlying a correlation plot is a correlation matrix, which we must calculate first. pandas makes this easy. We just need to select our columns of features and response values using the list we just created and call the .corr() method on these columns. As we calculate this, note that the type of correlation available to us in pandas is linear correlation, also known as Pearson correlation. Pearson correlation is used to measure the strength and direction (that is, positive or negative) of the linear relationship between two variables:

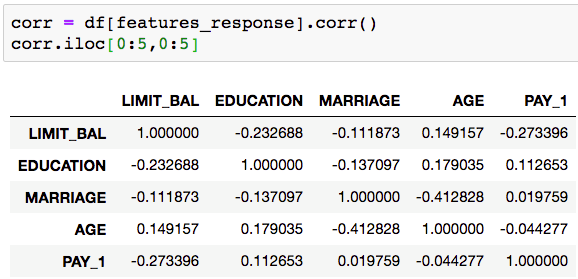


Figure 3.4: First five rows and columns of the correlation matrix

After creating the correlation matrix, notice that the row and column names are the same. Then, for each possible comparison between all pairs of features, as well as all features and the response, which we can't yet see here in the first five rows and columns, there is a number. This number is called the correlation between these two columns. All the correlations are between -1 and 1; a column has a correlation of 1 with itself (the diagonal of the correlation matrix), and there is repetition: each comparison appears twice since each column name from the original DataFrame appears as both a row and column in the correlation matrix. Before saying more about correlation, we'll use Seaborn to make a nice plot of it. Here is the plotting code, followed by the output (please see the notebook on GitHub for a color figure if you’re reading in black and white; it’s necessary here):

sns.heatmap(corr,

            xticklabels=corr.columns.values,

            yticklabels=corr.columns.values,

            center=0)

You should see the following output:

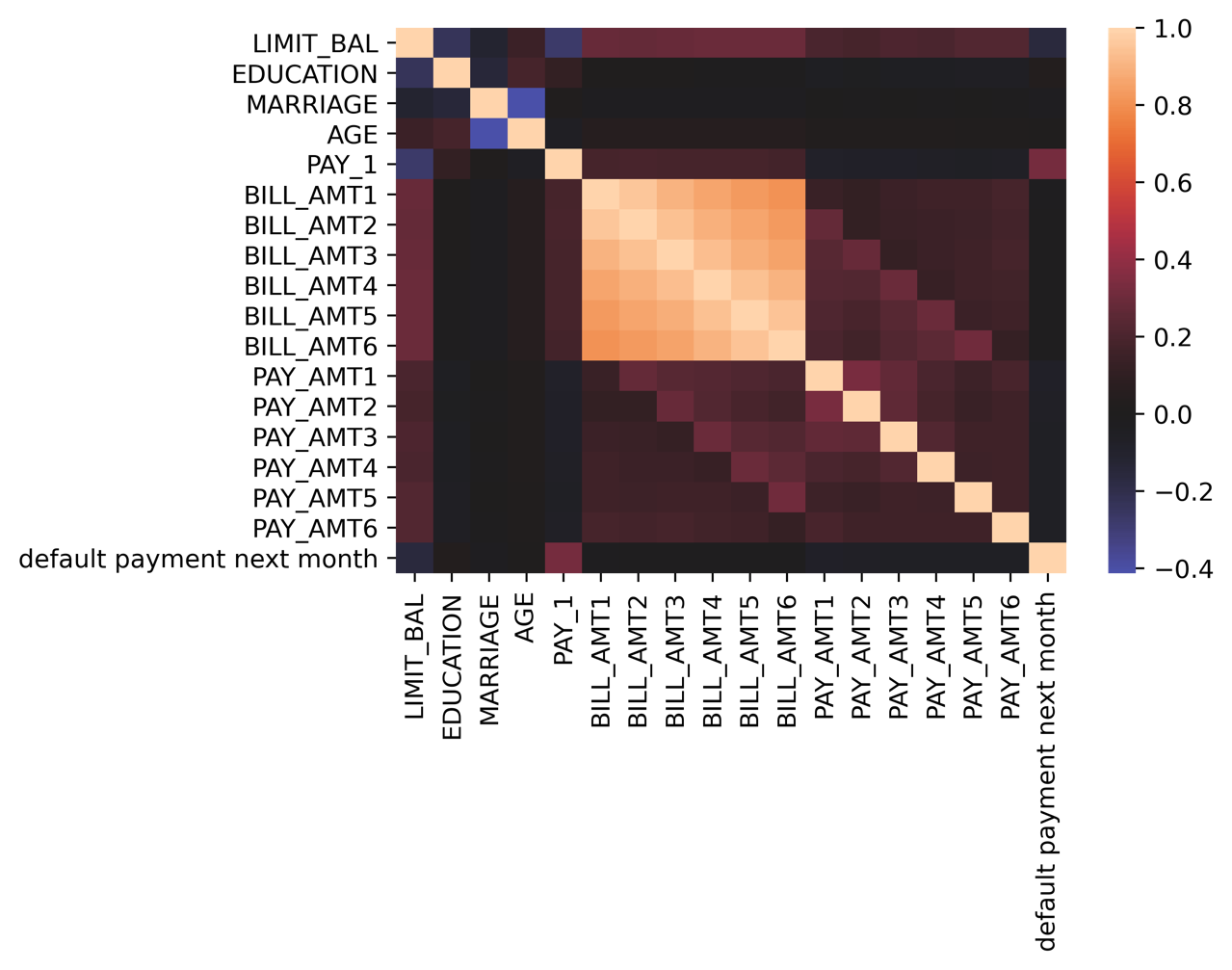


Figure 3.5: Heatmap of the correlation plot in Seaborn

The Seaborn heatmap feature makes an obvious visualization of the correlation matrix, according to the color scale on the right of Figure 3.5, which is called a colorbar. Notice that when calling sns.heatmap, in addition to the matrix, we supplied the tick labels for the x and y axes, which are the features and response names, and indicated that the center of the colorbar should be 0, so that positive and negative correlation are distinguishable as red and blue, respectively.

What does this plot tell us? At a high level, if two features, or a feature and the response, are highly correlated with each other, you can say there is a strong association between them. Features that are highly correlated to the response will be good features to use for prediction. This high correlation could be positive or negative; we'll explain the difference shortly.

To see the correlation with the response variable, we look along the bottom row, or equivalently, the last column. Here we see that the PAY\_1 feature is probably the most strongly correlated feature to the response variable. We can also see that a number of features are highly correlated to each other, in particular the BILL\_AMT features. We will talk in the next chapter about the importance of features that are correlated with each other; this is important to know about for certain models, such as logistic regression, that make assumptions about the correlations between features. For now, we make the observation that PAY\_1 is likely going to be one of the best, most predictive features for our model. The other feature that looks like it may be important is LIMIT\_BAL, which is negatively correlated. Depending on how astute your vision is, only these two really appear to be any color other than black (meaning 0 correlation) in the bottom row of Figure 3.5.

Mathematics of Linear Correlation

What is linear correlation, mathematically speaking? If you've taken basic statistics, you are likely familiar with linear correlation already. Linear correlation works very similarly to linear regression. For two columns, X and Y, linear correlation ρ (the lowercase Greek letter "rho") is defined as the following:

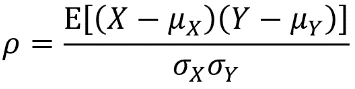


Figure 3.6: Linear correlation equation

This equation describes the expected value (E, which you can think of as the average) of the difference between the elements of X and their average, µx, multiplied by the difference between the corresponding elements of Y and their average, µy. The average for E is taken over pairs of X, Y values. You can imagine that if, when X is relatively large compared to its mean, µx, Y also tends to be similarly large, then the terms of the multiplication in the numerator will both tend to be positive, leading to a positive product and positive correlation after the expected value, E, is taken. Similarly, if Y tends to be small when X is small, both terms in the numerator will be negative and again lead to positive correlation. Conversely, if Y tends to decrease as X increases, they will have negative correlation. The denominator (the product of the standard deviations of X and Y) serves to normalize linear correlation to the scale of [-1, 1]. Because Pearson correlation is adjusted for the mean and standard deviation of the data, the actual values of the data are not as important as the relationship between X and Y. Stronger linear correlations are closer to 1 or -1. If there is no linear relation between X and Y, the correlation will be close to 0.

It's worth noting that, while it is regularly used in this context by data science practitioners, Pearson correlation is not strictly appropriate for a binary response variable, as we have in the case study problem. Technically speaking, among other restrictions, Pearson correlation is only valid for continuous data, such as the data we used for our linear regression exercise in Chapter 2, Introduction to Scikit-Learn and Model Evaluation. However, Pearson correlation can still accomplish the purpose of giving a quick idea of the potential usefulness of features. It is also conveniently available in software libraries such as pandas.

In data science in general, you will find that certain widely used techniques may be applied to data that violate their formal statistical assumptions. It is important to be aware of the formal assumptions underlying analytical methods. In fact, knowledge of these assumptions may be tested during interviews for data science jobs. However, in practice, as long as a technique can help us on our way to understanding the problem and finding an effective solution, it can still be a valuable tool.

That being said, linear correlation will not be an effective measure of the predictive power of all features. In particular, it only picks up on linear relationships. Shifting our focus momentarily to a hypothetical regression problem, have a look at the following examples and discuss what you expect the linear correlations to be. Notice that the values of the data on the xandy axes are not labeled; this is because the location (mean) and standard deviation (scale) of data does not affect the Pearson correlation, only the relationship between the variables, which can be discerned by plotting them together:

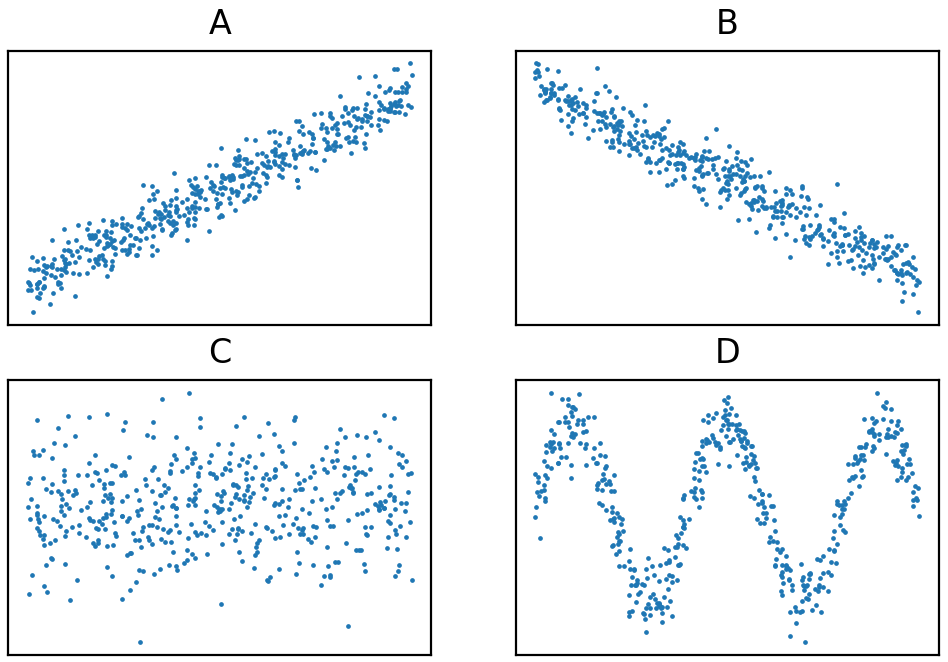


Figure 3.7: Scatter plots of the relationship between example variables

For examples A and B, the actual Pearson correlations of these datasets are 0.96 and -0.97, respectively, according to the formula given previously. From looking at the plots, it's pretty clear that a correlation close to 1 or -1 has provided useful insight into the relationship between these variables. For example C, the correlation is 0.06. A correlation closer to 0 looks like an effective indication of the lack of an association here: the value of Y doesn't really seem to have much to do with the value of X. However, in example D, there is clearly some relationship between the variables. But the linear correlation is actually lower than the previous example, at 0.02. Here, X and Y tend to "move together" over smaller scales, but this is averaged out over all samples when the linear correlation is calculated.

Ultimately, any summary statistic such as correlation that you may choose is only that: a summary. It could hide important details. For this reason, it is usually a good idea to visually examine the relationship between the features and response. This potentially takes up a lot of space on the page, so we won't demonstrate it here for all features in the case study. However, both pandas and Seaborn offer functions to create what's called a scatter plot matrix. A scatter plot matrix is similar to a correlation plot, but it actually shows all the data as a grid of scatter plots of all features and the response variable. This allows you to examine the data directly in a concise format. Since this could potentially be a lot of data and plots, you may need to downsample your data and look at a reduced number of features for the function to run efficiently.

Note

The code to generate the preceding plots can be found here: https://github.com/TrainingByPackt/Data-Science-Projects-With-Python-Second-Edition/tree/main/Chapter03/Reference

F-test

While Pearson correlation is theoretically valid for continuous response variables, the binary response variable for the case study data could be considered categorical data, with only two categories: 0 and 1. Among the different kinds of tests we can run, to see whether features are associated with a categorical response, is the ANOVA F-test, available in scikit-learn as f\_classif. ANOVA stands for analysis of variance. The ANOVA F-test can be contrasted with the regression F-test, which is very similar to Pearson correlation, also available in scikit-learn as f\_regression.

We will do an ANOVA F-test using the candidate features for the case study data in the following exercise. You will see that the output consists of F-statistics, as well as p-values. How can we interpret this output? We will focus on the p-value, for reasons that will become clear in the exercise. The p-value is a useful concept across a wide variety of statistical measures. For instance, although we didn't examine them, each of the Pearson correlations calculated for the preceding correlation matrix has a corresponding p-value. There is a similar concept of a p-value corresponding to linear regression coefficients, logistic regression coefficients, and other measures.

In the context of the F-test, the p-value answers the question: "For the samples in the positive class, how likely is it that the average value of this feature is the same as that of samples in the negative class?" If the data indicated that a feature has very different average values between the positive and negative classes, the following will be the case:

* It will be very unlikely that those average values are the same (low p-value).
* It will probably be a good feature in our model because it will help us discriminate between positive and negative classes.

Keep these points in mind during the following exercise.

Exercise 3.01: F-test and Univariate Feature Selection

In this exercise, we'll use the F-test to examine the relationship between the features and response variable. We will use this method to do what is called univariate feature selection: the practice of testing features one by one against the response variable, to see which ones have predictive power. Perform the following steps to complete the exercise:

Note

The Jupyter notebook for this exercise can be found here: https://github.com/TrainingByPackt/Data-Science-Projects-With-Python-Second-Edition/tree/main/Chapter03/Exercise01.

1. Our first step in doing the ANOVA F-test is to separate out the features and response as NumPy arrays, taking advantage of the list we created, as well as integer indexing in pandas:

X = df[features\_response].iloc[:,:-1].values

y = df[features\_response].iloc[:,-1].values

print(X.shape, y.shape)

The output should show the shapes of the features and response:

(26664, 17) (26664, )

There are 17 features, and both the features and response arrays have the same number of samples as expected.

1. Import the f\_classif function and feed in the features and response:

from sklearn.feature\_selection import f\_classif

[f\_stat, f\_p\_value] = f\_classif(X, y)

There are two outputs from f\_classif: the F-statistic and the p-value, for the comparison of each feature to the response variable. Let's create a new DataFrame containing the feature names and these outputs, to facilitate our inspection. One way to specify a new DataFrame is by using a dictionary, with key:value pairs of column names and the data to be contained in each column. We show the DataFrame sorted (ascending) on p-value.

1. Use this code to create a DataFrame of feature names, F-statistics, and p-values, and show it sorted on p-value:

f\_test\_df = pd.DataFrame({'Feature':features\_response[:-1],

                          'F statistic':f\_stat,

                          'p value':f\_p\_value})

f\_test\_df.sort\_values('p value')

The output should look like this:

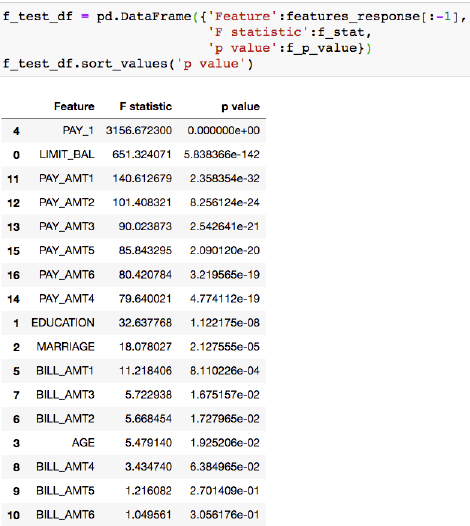


Figure 3.8: Results of the ANOVA F-test

Note that for every decrease in p-value, there is an increase in the F-statistic, so the information in these columns is identical in terms of ranking features.

The conclusions we can draw from the DataFrame of F-statistics and p-values are similar to what we observed in the correlation plot: PAY\_1 and LIMIT\_BAL appear to be the most useful features. They have the smallest p-values, indicating the average values of these features are significantly different between the positive and negative classes, and these features will help predict which class a sample belongs to.

In scikit-learn, measures such as the F-test help us perform univariate feature selection. This may be helpful if you have a very large number of features, many of which may be totally useless, and would like a quick way to get a short list of which ones might be most useful. For example, if we wanted to retrieve only the 20% of features with the highest F-statistics, we could do this easily with the SelectPercentile class. Also note there is a similar class for the selection of the top "k" features (where k is any number you specify), called SelectKBest. Here we demonstrate how to select the top 20%.

1. To select the top 20% of features according to the F-test, first import the SelectPercentile class:

from sklearn.feature\_selection import SelectPercentile

1. Instantiate an object of this class, indicating we'd like to use the same feature selection criteria, ANOVA F-test, that we've already been considering in this exercise, and that we'd like to select the top 20% of features:

selector = SelectPercentile(f\_classif, percentile=20)

1. Use the .fit method to fit the object on our features and response data, similar to how a model would be fit:

selector.fit(X, y)

The output should appear like this:

SelectPercentile(percentile=20)

There are several ways to access the selected features directly, which you may learn about in the scikit-learn documentation (that is, the .transform method, or in the same step as fitting with .fit\_transform). However, these methods will return NumPy arrays, which don't tell you the names of the features that were selected, just the values. For that, you can use the .get\_support method of the feature selector object, which will give you the column indices of the feature array that were selected.

1. Capture the indices of the selected features in an array named best\_feature\_ix:

best\_feature\_ix = selector.get\_support()

best\_feature\_ix

The output should appear as follows, indicating a logical index that can be used with an array of feature names, as well as values, assuming they're in the same order as the features array supplied to SelectPercentile:

array([ True, False, False, False, True, False, False, False, False,

False, False, True, True, False, False, False, False])

1. The feature names can be obtained using all but the last element (the name response variable) of our features\_response list by indexing with :-1:

features = features\_response[:-1]

1. Use the index array we created in Step 7 with a list comprehension and the features list, to find the selected feature names, as follows:

best\_features = [features[counter]

                 for counter in range(len(features))

                 if best\_feature\_ix[counter]]

best\_features

The output should be as follows:

['LIMIT\_BAL', 'PAY\_1', 'PAY\_AMT1', 'PAY\_AMT2']

In this code, the list comprehension has looped through the number of elements in the features array (len(features)) with the counter loop increment, using the best\_feature\_ix Boolean array, representing selected features, in the if statement to test whether each feature was selected and capturing the name if so.

The selected features agree with the top four rows of our DataFrame of F-test results, so the feature selection has worked as expected. While it's not strictly necessary to do things both ways, since they both lead to the same result, it's good to check your work, especially as you are learning new concepts. You should be aware that with convenient methods such as SelectPercentile, you don't get visibility of the F-statistics or p-values. However, in some situations, it may be more convenient to use these methods, as the p-values may not necessarily be important, outside of their utility in ranking features.

Finer Points of the F-test: Equivalence to the t-test for Two Classes and Cautions

When we use an F-test to look at the difference in means between just two groups, as we've done here for the binary classification problem of the case study, the test we are performing actually reduces to what's called a t-test. An F-test is extensible to three or more groups and so is useful for multiclass classification. A t-test just compares the means between two groups of samples, to see whether the difference in those means is statistically significant.

While the F-test served our purposes here of univariate feature selection, there are a few cautions to keep in mind. Going back to the concept of formal statistical assumptions, for the F-test these include that the data is normally distributed. We have not checked this. Also, in comparing the same response variable, y, to many potential features from the matrix, X, we have performed what is known in statistics as multiple comparisons. In short, this means that by examining multiple features in comparison to the same response over and over, the odds increase that we'll find what we think is a "good feature" just by random chance. However, such features may not generalize to new data. There are statistical corrections for multiple comparisons that amount to adjusting the p-values to account for this.

Even if we have not followed all the statistical rules that go along with these methods, we can still get useful results from them. The multiple comparisons correction is more of a concern when p-values are the ultimate quantity of interest, for example, when making statistical inferences. Here, p-values are just a means to an end of ranking the feature list. The order of this ranking would not change if the p-values were corrected for multiple comparisons.

In addition to knowing which features are likely to be useful for modeling, it is good to have a deeper understanding of the important features. Consequently, we will do a detailed graphical exploration of these in the next exercise. We will also look at other methods for feature selection later that don't make the same assumptions as those we've introduced here and are more directly integrated with the predictive models that we will build.

Hypotheses and Next Steps

According to our univariate feature exploration, the feature with the strongest association with the response variable is PAY\_1. Does this make sense? What is the interpretation of PAY\_1? PAY\_1 is the payment status of the account, in the most recent month. As we learned in the initial data exploration, there are some values that indicate that the account was in good standing: -2 means no account usage, -1 means balance paid in full, and 0 means at least the minimum payment was made. On the other hand, positive integer values indicate a delay of payment by that many months. Accounts with delayed payments last month were accounts that could be considered in default. This means that, essentially, this feature captures historical values of the response variable. Features such as this are extremely important as one of the best predictors for just about any machine learning problem is historical data on the same thing you are trying to predict (that is, the response variable). This should make sense: people who have defaulted before are probably at the highest risk of defaulting again.

How about LIMIT\_BAL, the credit limit of accounts? Thinking about how credit limits are assigned, it is likely that our client has assessed how risky a borrower is when deciding their credit limit. Riskier clients should be given lower limits, so the creditor is less exposed. Therefore, we may expect to see a higher probability of default for accounts with lower values for LIMIT\_BAL.

What have we learned from our univariate feature selection exercise? We have an idea of what the most important features in our model are likely to be. And, from the correlation matrix, we have some idea of how they are related to the response variable. However, knowing the limitations of the tests we used, it is a good idea to visualize these features for a closer look at the relationship between the features and response variable. We have also started to develop hypotheses about these features: why do we think they are important? Now, by visualizing the relationships between the features and the response variable, we can determine whether our ideas are compatible with what we can see in the data.

Such hypotheses and visualizations are often a key part of presenting your results to a client, who may be interested in how a model works, not just the fact that it does work.

Exercise 3.02: Visualizing the Relationship between the Features and Response Variable

In this exercise, you will further your knowledge of plotting functions from Matplotlib that you used earlier in this book. You'll learn how to customize graphics to better answer specific questions with the data. As you pursue these analyses, you will create insightful visualizations of how the PAY\_1 and LIMIT\_BAL features relate to the response variable, which may possibly provide support for the hypotheses you formed about these features. This will be done by becoming more familiar with the Matplotlib Application Programming Interface (API), in other words, the syntax you use to interact with Matplotlib. 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 here: https://github.com/TrainingByPackt/Data-Science-Projects-With-Python-Second-Edition/tree/main/Chapter03/Exercise02.

1. Calculate a baseline for the response variable of the default rate across the whole dataset using pandas' .mean():

overall\_default\_rate = df['default payment next month'].mean()

overall\_default\_rate

The output of this should be the following:

0.2217971797179718

What would be a good way to visualize default rates for different values of the PAY\_1 feature?

Recall our observation that this feature is sort of like a hybrid categorical and numerical feature. We'll choose to plot it in a way that is typical for categorical features, due to the relatively small number of unique values. In Chapter 1, Data Exploration and Cleaning, we did value\_counts of this feature as part of data exploration, then later we learned about groupby/mean when looking at the EDUCATION feature. groupby/mean would be a good way to visualize the default rate again here, for different payment statuses.

1. Use this code to create a groupby/mean aggregation:

group\_by\_pay\_mean\_y = df.groupby('PAY\_1').agg(

{'default payment next month':np.mean})

group\_by\_pay\_mean\_y

The output should look as follows:

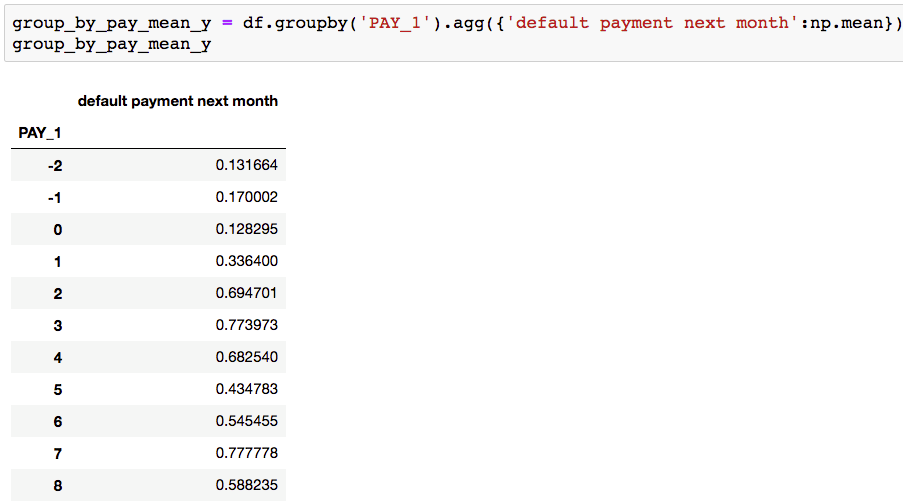


Figure 3.9: Mean of the response variable by groups of the PAY\_1 feature

Looking at these values, you may already be able to discern the trend. Let's go straight to plotting them. We'll take it step by step and introduce some new concepts. You should put all the code from Steps 3 through 6 in a single code cell.

In Matplotlib, every plot exists on an axes, and within a figure window. By creating objects for axes and figure, you can directly access and change their properties, including axis labels and other kinds of annotation on the axes.

1. Create an axes object in a variable also called axes, using the following code:

axes = plt.axes()

1. Plot the overall default rate as a red horizontal line.

Matplotlib makes this easy; you just have to indicate the y intercept of this line with the axhline function. Notice that instead of calling this function from plt, now we are calling it as a method on our axes object:

axes.axhline(overall\_default\_rate, color='red')

Now, over this line, we want to plot the default rate within each group of PAY\_1 values.

1. Use the plot method of the DataFrame of grouped data we created. Specify to include an 'x' marker along the line plot, to not have a legend instance, which we'll create later, and that the parent axes of this plot should be the axes we are already working with (otherwise, pandas would erase what was already there and create new axes):

group\_by\_pay\_mean\_y.plot(marker='x', legend=False, ax=axes)

This is all the data we want to plot.

1. Set the y-axis label and create a legend instance (there are many possible options for controlling the legend appearance, but a simple way is to provide a list of strings, indicating the labels for the graphical elements in the order they were added to the axes):

axes.set\_ylabel('Proportion of credit defaults')

axes.legend(['Entire dataset', 'Groups of PAY\_1'])

1. Executing all the code from Steps 3 through 6 in a single code cell should result in the following plot:

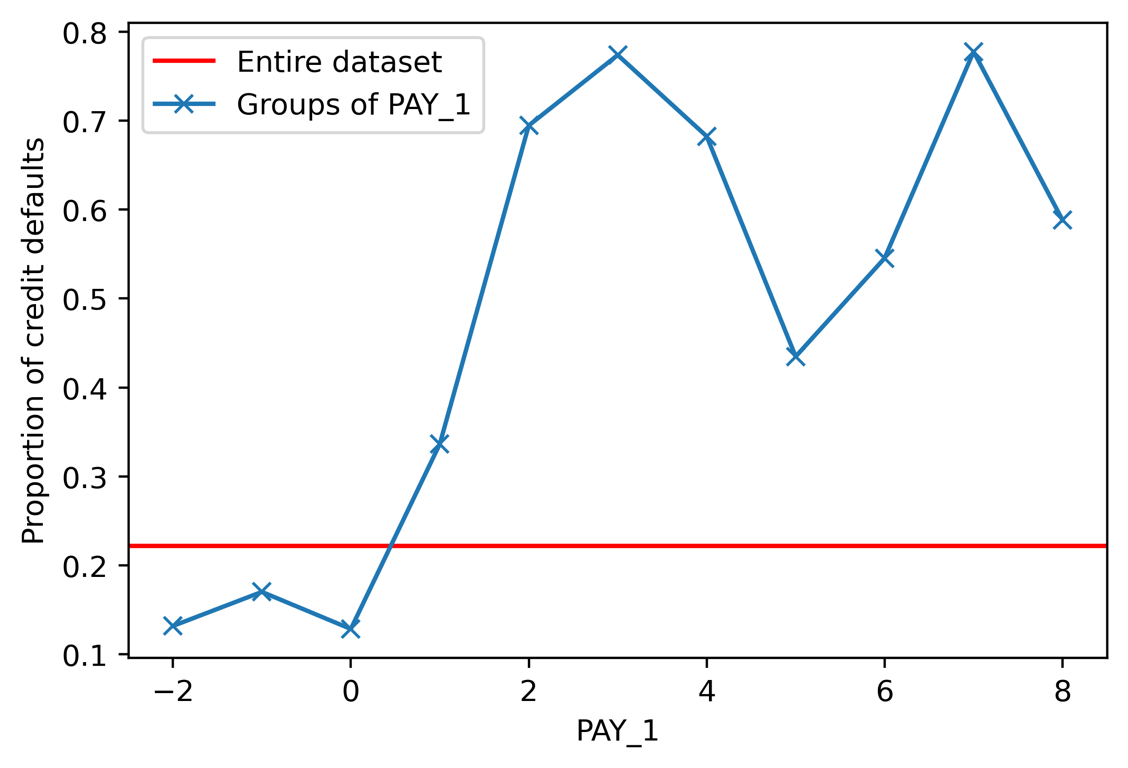


Figure 3.10: Credit default rates across the dataset

Our visualization of payment statuses has revealed a clear, and probably expected, story: those who defaulted before are in fact more likely to default again. The default rate of accounts in good standing is well below the overall default rate, which we know from before is about 22%. However, over 30% of the accounts that were in default last month will be in default again next month, according to this. This is a good visual to share with our business partner as it shows the effect of what may be one of the most important features in our model.

Now we turn our attention to the feature ranked as having the second strongest association with the target variable: LIMIT\_BAL. This is a numerical feature with many unique values. A good way to visualize features such as this, for a classification problem, is to plot multiple histograms on the same axis, with different colors for the different classes. As a way to separate the classes, we can index them from the DataFrame using logical arrays.

1. Use this code to create logical masks for positive and negative samples:

pos\_mask = y == 1

neg\_mask = y == 0

To create our dual histogram plot, we'll make another axes object, then call the .hist method on it twice for the positive and negative class histograms. We supply a few additional keyword arguments: the first histogram will have black edges and white bars, while the second will use alpha to create transparency, so we can see both histograms in the places they overlap. Once we have the histograms, we rotate the x-axis tick labels to make them more legible and create several other annotations that should be self-explanatory.

1. Use the following code to create the dual histogram plot with the aforementioned properties:

axes = plt.axes()

axes.hist(df.loc[neg\_mask, 'LIMIT\_BAL'],\

          edgecolor='black', color='white')

axes.hist(df.loc[pos\_mask, 'LIMIT\_BAL'],\

          alpha=0.5, edgecolor=None, color='black')

axes.tick\_params(axis='x', labelrotation=45)

axes.set\_xlabel('Credit limit (NT$)')

axes.set\_ylabel('Number of accounts')

axes.legend(['Not defaulted', 'Defaulted'])

axes.set\_title('Credit limits by response variable')

The plot should appear like this:

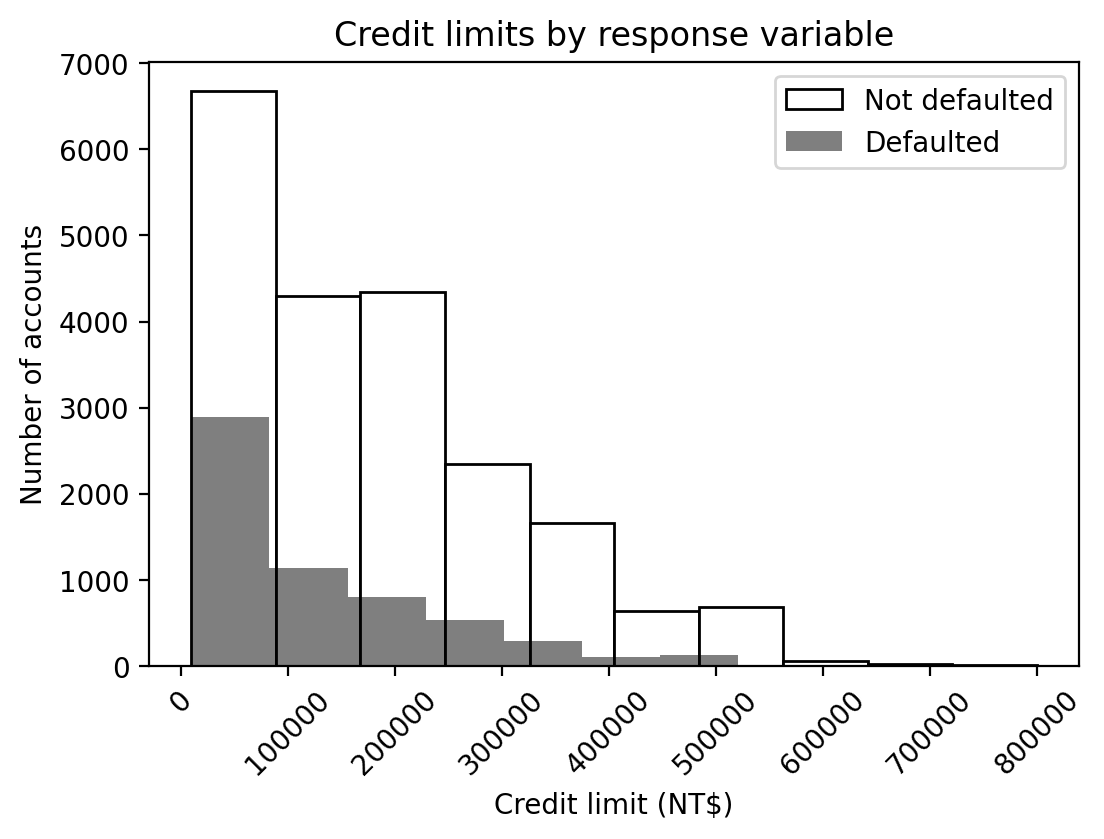


Figure 3.11: Dual histograms of credit limits

While this plot has accomplished all the formatting we wished to present, it's not quite as interpretable as it could be. What we hope to gain from looking at it is some knowledge of how the credit limit may be a good way to distinguish between accounts that default and those that do not. However, the primary visual takeaway here is that the transparent histogram is bigger than the gray one. This is due to the fact that fewer accounts default than don't default. We already know this from examining the class fractions.

It would be more informative to show something about how the shapes of these histograms are different, not just their sizes. To emphasize this, we can make the total plotted area of the two histograms the same, by normalizing them. Matplotlib provides a keyword argument that makes this easy, creating what might be considered an empirical version of a probability mass function. This means that the integral or area contained within each histogram will be equal to 1 after normalization, since probabilities sum to 1.

After some experimentation, we decide to make a histogram with 16 bins. Since the maximum credit limit is NT$800,000, we use range with an increment of NT$50,000.

Note

You can find the maximum credit limit by running df['LIMIT\_BAL'].max()

1. Create and display the histogram bin edges with this code:

bin\_edges = list(range(0,850000,50000))

print(bin\_edges)

The output should be as follows:

[0, 50000, 100000, 150000, 200000, 250000, 300000, 350000, 40000, 450000,

500000, 550000, 600000, 650000, 700000, 750000, 800000]

The plotting code for the normalized histograms is similar to before, with a few key changes: the use of the bins keyword to define bin edge locations, density=True to normalize the histograms, and changes to the plot annotations. The most complex part is that we need to adjust the y-axis tick labels, so that the heights of the histogram bins have the interpretation of proportions, which is more intuitive than the default output.

Y-axis tick labels are the text labels displayed next to the ticks on the y axis and are usually simply the values of the ticks at those locations. However, you are able to manually change this if you want.

Note

According to the Matplotlib documentation, for a normalized histogram, the bin heights are calculated by "dividing the count by the number of observations times the bin width" (<https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html>). So, we need to multiply the y-axis tick labels by the bin width of NT$50,000, for the bin heights to represent the proportion of the total number of samples in each bin. Notice the two lines where we get the tick locations of the y-axis, then set the labels to a modified version. The rounding to two decimal places with np.round is needed due to slight errors of floating-point arithmetic.

1. Run this code to produce normalized histograms:

mpl.rcParams['figure.dpi'] = 400

axes = plt.axes()

axes.hist(

df.loc[neg\_mask, 'LIMIT\_BAL'],

bins=bin\_edges, density=True,

edgecolor='black', color='white')

axes.hist(

df.loc[pos\_mask, 'LIMIT\_BAL'],

bins=bin\_edges, density=True, alpha=0.5,

edgecolor=None, color='black')

axes.tick\_params(axis='x', labelrotation=45)

axes.set\_xlabel('Credit limit (NT$)')

axes.set\_ylabel('Proportion of accounts')

y\_ticks = axes.get\_yticks()

axes.set\_yticklabels(np.round(y\_ticks\*50000,2))

axes.legend(['Not defaulted', 'Defaulted'])

axes.set\_title('Normalized distributions of '\

               'credit limits by response variable')

The plot should look like this:

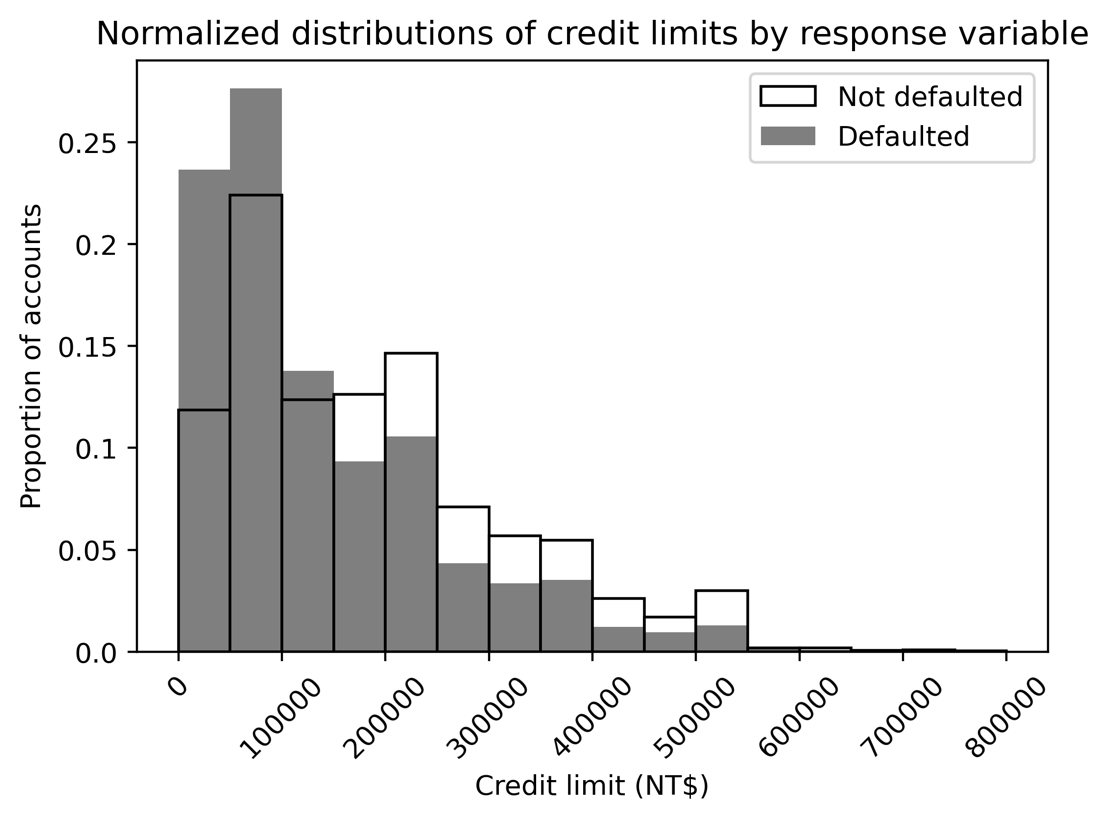


Figure 3.12: Normalized dual histograms

You can see that plots in Matplotlib are highly customizable. In order to view all the different things you can get from and set on Matplotlib axes, have a look here: https://matplotlib.org/stable/api/axes\_api.html.

What can we learn from this plot? It looks like the accounts that default tend to have a higher proportion of lower credit limits. Accounts with credit limits less than NT$150,000 are relatively more likely to default, while the opposite is true for accounts with limits higher than this. We should ask ourselves, does this make sense? Our hypothesis was that the client would give riskier accounts lower limits. This intuition is compatible with the higher proportions of defaulters with lower credit limits that we observed here.

Depending on how the model building goes, if the features we examined in this exercise turn out to be important for predictive modeling as we expect, it would be good to show these graphs to our client, as part of a presentation of our work. This would give the client insight into how the model works, as well as insights into their data.

A key learning from this section is that effective visual presentations take substantial time to produce. It is good to budget some time in your project workflow for this. Convincing visuals are worth the effort since they should be able to quickly and effectively communicate important findings to the client. They are usually a better choice than adding lots of text to the materials that you create. Visual communication of quantitative concepts is a core data science skill.

Univariate Feature Selection: What It Does and Doesn't Do

In this chapter, we have learned techniques for going through features one by one to see whether they have predictive power. This is a good first step, and if you already have features that are very predictive of the outcome variable, you may not need to spend much more time considering features before modeling. However, there are drawbacks to univariate feature selection. In particular, it does not consider the interactions between features. For example, what if the credit default rate is very high specifically for people with both a certain education level and a certain range of credit limit?

Also, with the methods we used here, only the linear effects of features are captured. If a feature is more predictive when it's undergone some type of transformation, such as a polynomial or logarithmic transformation, or binning (discretization), linear techniques of univariate feature selection may not be effective. Interactions and transformations are examples of feature engineering, or creating new features, in these cases from existing features. The shortcomings of linear feature selection methods can be remedied by non-linear modeling techniques including decision trees and methods based on them, which we will examine later. But there is still value in looking for simple relationships that can be found by linear methods for univariate feature selection, and it is quick to do.

Understanding Logistic Regression and the Sigmoid Function Using Function Syntax in Python

In this section, we will open the “black box” of logistic regression all the way: we will gain a comprehensive understanding of how it works. We’ll start off by introducing a new programming concept: functions. At the same time, we'll learn about a mathematical function, the sigmoid function, which plays a key role in logistic regression.

In the most basic sense, a function in computer programming is a piece of code that takes inputs and produces outputs. You have been using functions throughout the book: functions that were written by someone else. Any time that you use syntax such as this: output = do\_something\_to(input), you have used a function. For example, NumPy has a function you can use to calculate the mean of the input:

np.mean([1, 2, 3, 4, 5])

3.0

Functions abstract away the operations being performed so that, in our example, you don't need to see all the lines of code that it takes to calculate a mean, every time you need to do this. For many common mathematical functions, there are already pre-defined versions available in packages such as NumPy. You do not need to "reinvent the wheel." The implementations in popular packages are likely popular for a reason: people have spent time thinking about how to create them in the most efficient way. So, it would be wise to use them. However, since all the packages we are using are open source, if you are interested in seeing how the functions in the libraries we use are implemented, you are able to look at the code within any of them.

Now, for the sake of illustration, let's learn Python function syntax by writing our own function for the arithmetic mean. Function syntax in Python is similar to for or if blocks, in that the body of a function is indented and the declaration of the function is followed by a colon. Here is the code for a function to compute the mean:

def my\_mean(input\_argument):

    output = sum(input\_argument)/len(input\_argument)

    return(output)

After you execute the code cell with this definition, the function is available to you in other code cells in the notebook. Take the following example:

my\_mean([1, 2, 3, 4, 5])

3.0

The first part of defining a function, as shown here, is to start a line of code with def, followed by a space, followed by the name you'd like to call the function. After this come parentheses, inside which the names of the parameters of the function are specified. Parameters are names of the input variables, where these names are internal to the body of the function: the variable names defined as parameters are available within the function when it is called (used), but not outside the function. There can be more than one parameter; they would be comma-separated. After the parentheses comes a colon.

The body of the function is indented and can contain any code that operates on the inputs. Once these operations are done, the last line should start with return and contain the output variable(s), comma-separated if there is more than one. We are leaving out many fine points in this very simple introduction to functions, but those are the essential parts you need to get started.

The power of a function comes when you use it. Notice how after we define the function, in a separate code block we can call it by the name we've given it, and it operates on whatever inputs we pass it. It's as if we've copied and pasted all the code to this new location. But it looks much nicer than actually doing that. And if you are going to use the same code many times, a function can greatly reduce the overall length of your code.

As a brief additional note, you can optionally specify the inputs using the parameter names explicitly, which can be clearer when there are many inputs:

my\_mean(input\_argument=[1, 2, 3])

2.0

Now that we're familiar with the basics of Python functions, we are going to consider a mathematical function that's important to logistic regression, called sigmoid. This function may also be called the logistic function. The definition of sigmoid is as follows:

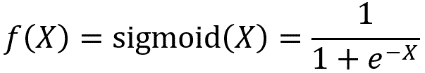


Figure 3.13: The sigmoid function

We will break down the different parts of this function. As you can see, the sigmoid function involves the irrational number e, which is also known as the base of the natural logarithm, in contrast to the base-10 logarithms we used earlier for data exploration. In order to compute e-X using Python, we don't actually need to perform the exponentiation manually. NumPy has a convenient function, exp, that takes e to the input exponent automatically. If you look at the documentation, you will see this process is called taking the "exponential," which sounds vague. But it is assumed to be understood that the base of the exponent is e in this case. In general, if you want to take an exponent in Python, such as 23 ("two to the third power"), the syntax is two asterisks: 2\*\*3, which equals 8, for example.

Consider how inputs may be passed to the np.exp function. Since NumPy's implementation is vectorized, this function can take individual numbers as well as arrays or matrices as input. To illustrate individual arguments, we compute the exponential of 1, which shows the approximate value of e, as well as e0, which of course equals 1, as does the zeroth power of any base:

np.exp(1)

2.718281828459045

np.exp(0)

1.0

To illustrate the vectorized implementation of np.exp, we create an array of numbers using NumPy's linspace function. This function takes as input the starting and stopping points of a range, both inclusive, and the number of values you'd like within that range, to create an array of that many linearly spaced values. This function performs a somewhat similar role to Python's range, but can also produce decimal values:

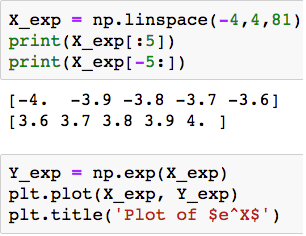


Figure 3.14: Using np.linspace to make an array

Since np.exp is vectorized, it will compute the exponential of the whole array at once, in an efficient manner. Here is the code with output, to calculate the exponential of our X\_exp array and examine the first five values:

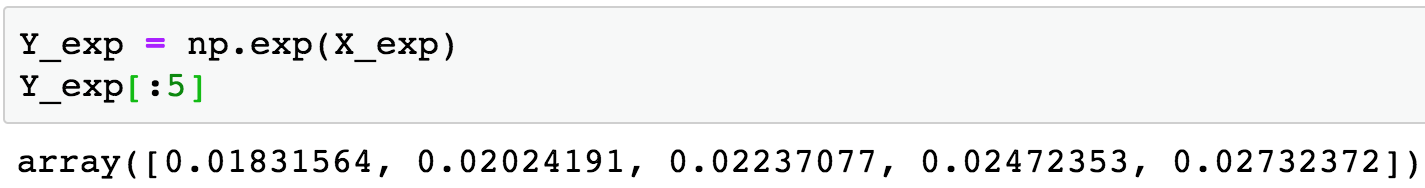


Figure 3.15: NumPy's exp function

Exercise 3.03: Plotting the Sigmoid Function

In this exercise, we will use X\_exp and Y\_exp, created previously, to make a plot of what the exponential function looks like over the interval [-4, 4]. You need to have run all the code in Figures 3.14 and 3.15 to have these variables available for this exercise. Then we will define a function for the sigmoid, create a plot of that, and consider how it is related to the exponential function. 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 here: <https://github.com/TrainingByPackt/Data-Science-Projects-With-Python-Second-Edition/tree/main/Chapter03/Exercise03>.

1. Use this code to plot the exponential function:

plt.plot(X\_exp, Y\_exp)

plt.title('Plot of $e^X$')

The plot should look like this:

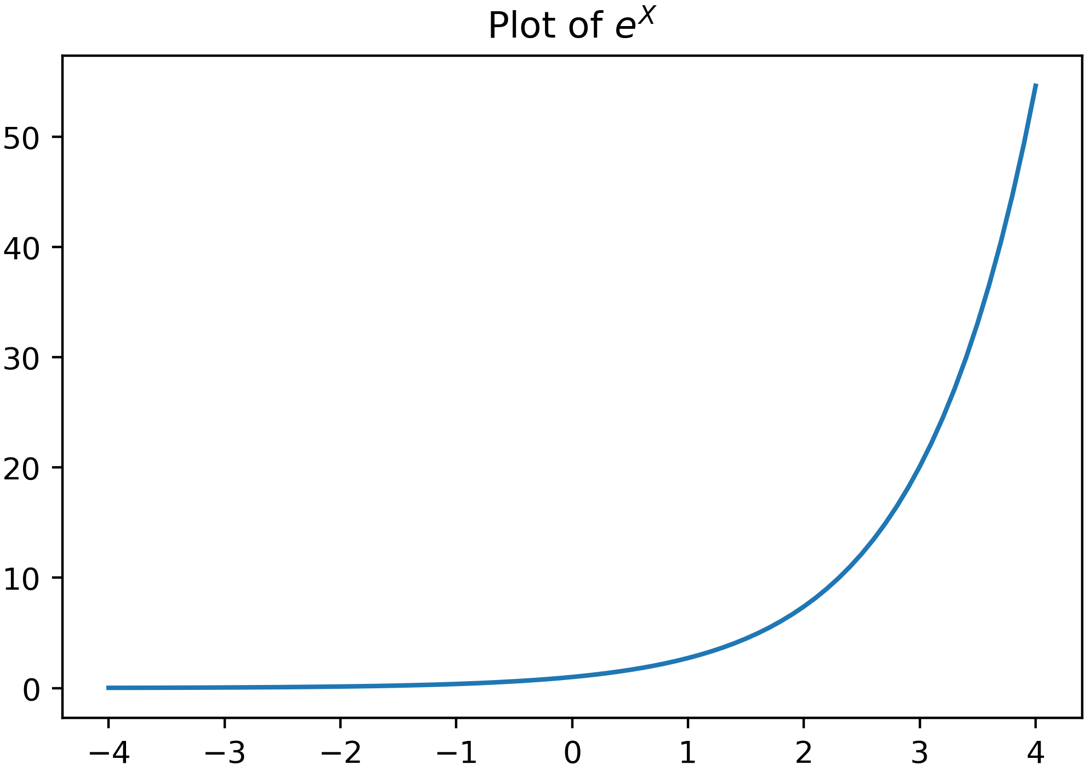


Figure 3.16: Plotting the exponential function

Notice that in titling the plot, we've taken advantage of a kind of syntax called LaTeX, which enables the formatting of mathematical notation. We won't go into the details of LaTeX here, but suffice to say that it is very flexible. Note that enclosing part of the title string in dollar signs causes it to be rendered using LaTeX, and that superscript can be created using ^.

Also note in Figure 3.16 that many points spaced close together create the appearance of a smooth curve, but in fact, it is a graph of discrete points connected by line segments.

What can we observe about the exponential function?

It is never negative: as X approaches negative infinity, Y approaches 0.

As X increases, Y increases slowly at first, but very quickly "blows up." This is what is meant when people say "exponential growth" to signify a rapid increase.

How can you think about the sigmoid in terms of the exponential?

First, the sigmoid involves e-X, as opposed to eX. The graph of e-X is just the reflection of eX about the y axis. This can be plotted easily and annotated using curly braces for multiple-character superscript in the plot title.

1. Run this code to see the plot of e-X:

Y\_exp = np.exp(-X\_exp)

plt.plot(X\_exp, Y\_exp)

plt.title('Plot of $e^{-X}$')

The output should appear like this:

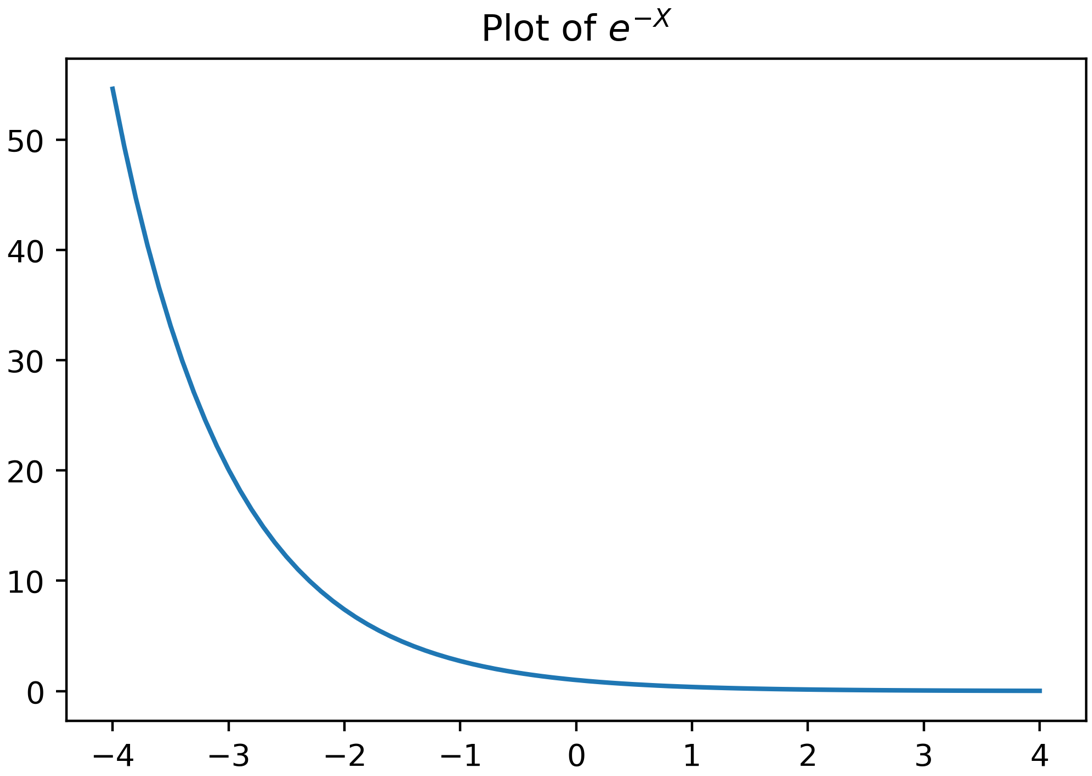


Figure 3.17: Plot of exp(-X)

Now, in the sigmoid function, e-X is in the denominator, with 1 added to it. The numerator is 1. So, what happens to the sigmoid as X approaches negative infinity? We know that e-X "blows up," becoming very large. Overall, the denominator becomes very large and the fraction approaches 0. What about when X increases toward positive infinity? We can see that e-X becomes very close to 0. So, in this case, the sigmoid function would be approximately 1/1 = 1. This should give you an intuition that the sigmoid function stays between 0 and 1. Let's now implement a sigmoid function in Python and use it to create a plot to see how reality matches this intuition.

1. Define a sigmoid function like this:

def sigmoid(X):

    Y = 1 / (1 + np.exp(-X))

    return Y

1. Make a larger range of x values to plot over and plot the sigmoid. Use this code:

X\_sig = np.linspace(-7,7,141)

Y\_sig = sigmoid(X\_sig)

plt.plot(X\_sig,Y\_sig)

plt.yticks(np.linspace(0,1,11))

plt.grid()

plt.title('The sigmoid function')

The plot should look like this:

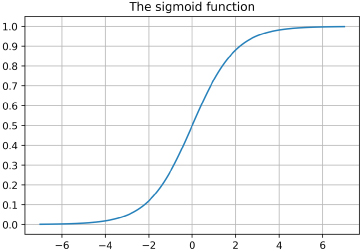


Figure 3.18: A sigmoid function plot

This plot matches what we expected. Further, we can see that sigmoid(0) = 0.5. What is special about the sigmoid function? The output of this function is strictly bounded between 0 and 1. This is a good property for a function that should predict probabilities, which are also required to be between 0 and 1. Technically, probabilities can be exactly equal to 0 and 1, while the sigmoid never is. But the sigmoid can be close enough that this is not a practical limitation.

Recall that we described logistic regression as producing predicted probabilities of class membership, as opposed to directly predicting class membership. This enables a more flexible implementation of logistic regression, allowing the selection of the threshold probability. The sigmoid function is the source of these predicted probabilities. Shortly, we will see how the different features are used in the calculation of the predicted probabilities.

Scope of Functions

As you begin to use functions, you should develop an awareness of the concept of scope. Notice that when we wrote the sigmoid function, we created a variable, Y, inside the function. Variables created inside functions are different from those created outside functions. They are effectively created and destroyed within the function itself when it is called. These variables are said to be local in scope: local to the function. If you have been running all the code as written in this chapter in a single notebook in sequence, notice that you are not able to access the Y variable after using the sigmoid function:

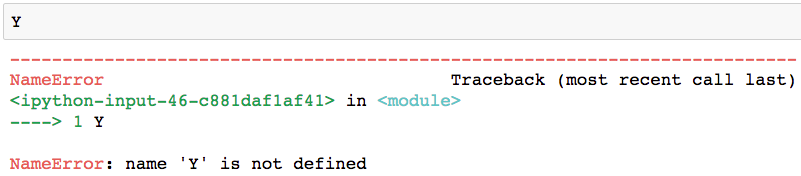


Figure 3.19: The Y variable not in the scope of the notebook

The Y variable is not in the global scope of the notebook. However, global variables created outside of functions are available within the local scope of functions, even if they are not inputted as parameters to the function. Here we demonstrate creating a variable outside of a function, which is global in scope, and then accessing it within a function. The function actually doesn't take any parameters at all, but as you can see, it can work with the value of the global variable to create an output:

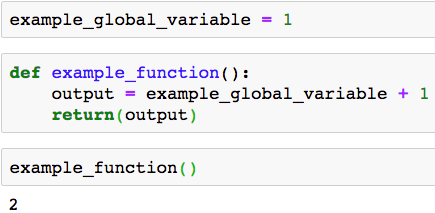


Figure 3.20: Global variable available within the local scope of the function

Note

More details on scope

The scope of variables can potentially be confusing but is good to know when you start making more advanced use of functions. While this knowledge isn't required for the book, you may wish to get a more in-depth perspective on variable scope in Python here: <https://nbviewer.jupyter.org/github/rasbt/python_reference/blob/master/tutorials/scope_resolution_legb_rule.ipynb>.

Sigmoid curves in scientific applications

Besides being fundamental to logistic regression, sigmoid curves are used in a variety of applications. In biology, they can be used to describe the growth of an organism, which starts slowly, then has a rapid phase, followed by a smooth tapering off as the final size is reached. Sigmoids can also be used to describe population growth, which has a similar trajectory, increasing rapidly but then slowing as the carrying capacity of the environment is reached.

Why Is Logistic Regression Considered a Linear Model?

We mentioned previously that logistic regression is considered a linear model, while we were exploring whether the relationship between features and response resembled a linear relationship. Recall that we plotted groupby/mean of the EDUCATION feature in Chapter 1, Data Exploration and Cleaning, as well as for the PAY\_1 feature in this chapter, to see whether the default rates across values of these features exhibited a linear trend. While this is a good way to get a quick approximation of how "linear or not" these features may be, here we formalize the notion of why logistic regression is a linear model.

A model is considered linear if the transformation of features that is used to calculate the prediction is a linear combination of the features. The possibilities for a linear combination are that each feature can be multiplied by a numerical constant, these terms can be added together, and an additional constant can be added. For example, in a simple model with two features, X1 and X2, a linear combination would take the following form:



Figure 3.21: Linear combination of X1 and X2

The constants 𝜃i can be any number, positive, negative, or zero, for i = 0, 1, and 2 (although if a coefficient is 0, this removes a feature from the linear combination). A familiar example of a linear transformation of one variable is a straight line with the equation y = mx + b. In this case, 𝜃o = b and 𝜃1 = m. 𝜃o is called the intercept of a linear combination, which should be familiar from algebra.

What kinds of things are "not allowed" in linear transformations? Any other mathematical expressions besides what was just described, such as the following:

* Multiplying a feature by itself; for example, X12 or X13. These are called polynomial terms.
* Multiplying features together; for example, X1X2. These are called interactions.
* Applying non-linear transformations to features; for example, log and square root.
* Other complex mathematical functions.
* "If then" types of statements. For example, "if X1 > a, then y = b."

However, while these transformations are not part of the basic formulation of a linear combination, they could be added to a linear model by engineering features, for example, defining a new feature, X3 = X12.

Earlier, we learned that the predictions of logistic regression, which take the form of probabilities, are made using the sigmoid function. Taking another look here, we see that this function is clearly non-linear:



Figure 3.22: Non-linear sigmoid function

Why, then, is logistic regression considered a linear model? It turns out that the answer to this question lies in a different formulation of the sigmoid equation, called the logit function. We can derive the logit function by solving the sigmoid function for X; in other words, finding the inverse of the sigmoid function. First, we set the sigmoid equal to p, which we interpret as the probability of observing the positive class, then solve for X as shown in the following:

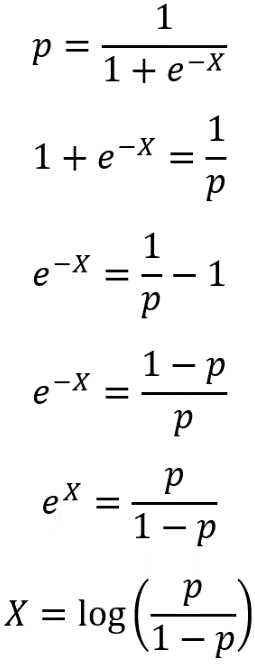


Figure 3.23: Solving for X

Here, we've used some laws of exponents and logs to solve for X. You may also see logit expressed as follows:

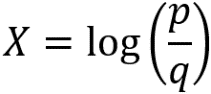


Figure 3.24: The logit function

In this expression, the probability of failure, q, is expressed in terms of the probability of success, p; q = 1 - p, because probabilities sum to 1. Even though in our case, credit default would probably be considered a failure in the sense of real-world outcomes, the positive outcome (response variable = 1 in a binary problem) is conventionally considered "success" in mathematical terminology. The logit function is also called the log odds, because it is the natural logarithm of the odds ratio, p/q. Odds ratios may be familiar from the world of gambling, via phrases such as "the odds are 2 to 1 that team a will defeat team b."

In general, what we've called capital X in these manipulations can stand for a linear combination of all the features. For example, this would be X = 𝜃o + 𝜃1X1 + 𝜃2X2 in our simple case of two features. Logistic regression is considered a linear model because the features included in X are, in fact, only subject to a linear combination when the response variable is considered to be the log odds. This is an alternative way of formulating the problem, as compared to the sigmoid equation.

Putting the pieces together, the features X1, X2,…, X~~j~~ look like this in the sigmoid equation version of logistic regression:

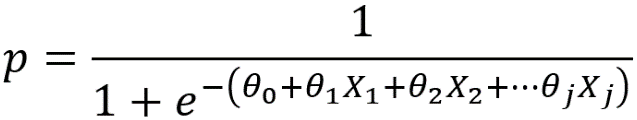


Figure 3.25: Sigmoid version of logistic regression

But they look like this in the log odds version, which is why logistic regression is called a linear model:

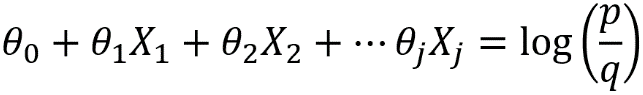


Figure 3.26: Log odds version of logistic regression

Because of this way of looking at logistic regression, ideally, the features of a logistic regression model would be linear in the log odds of the response variable. We will see what is meant by this in the following exercise.

Logistic regression is part of a broader class of statistical models called Generalized Linear Models (GLMs). GLMs are connected to the fundamental concept of ordinary linear regression, which may have one feature (that is, the line of best fit, y = mx + b, for a single feature, x) or more than one in multiple linear regression. The mathematical connection between GLMs and linear regression is the link function. The link function of logistic regression is the logit function we just learned about.

Exercise 3.04: Examining the Appropriateness of Features for Logistic Regression

In Exercise 3.02, Visualizing the Relationship between the Features and Response Variable, we plotted a groupby/mean of what might be the most important feature of the model, according to our exploration so far: the PAY\_1 feature. By grouping samples by the values of PAY\_1, and then looking at the mean of the response variable, we are effectively looking at the probability, p, of default within each of these groups.

In this exercise, we will evaluate the appropriateness of PAY\_1 for logistic regression. We will do this by examining the log odds of default within these groups to see whether the response variable is linear in the log odds, as logistic regression formally assumes. 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 here: https://github.com/TrainingByPackt/Data-Science-Projects-With-Python-Second-Edition/tree/main/Chapter03/Exercise04.

1. Confirm you still have access to the variables from Exercise 3.02, Visualizing the Relationship between the Features and Response Variable, in your notebook by reviewing the DataFrame of the average value of the response variable for different values of PAY\_1 with this code:

group\_by\_pay\_mean\_y

The output should be as follows:

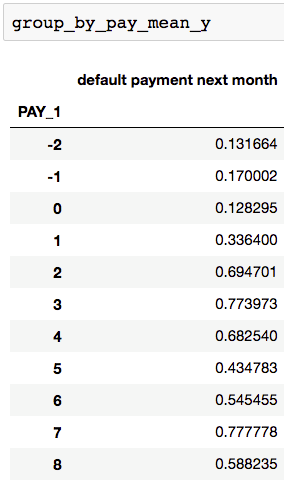


Figure 3.27: Rates of default within groups of PAY\_1 values as probabilities of default

1. Extract the mean values of the response variable from these groups and put them in a variable, p, representing the probability of default:

p = group\_by\_pay\_mean\_y['default payment next month'].values

1. Create a probability, q, of not defaulting. Since there are only two possible outcomes in this binary problem, and probabilities of all outcomes always sum to 1, it is easy to calculate q. Also print the values of p and q to confirm:

q = 1-p

print(p)

print(q)

The output should be as follows:

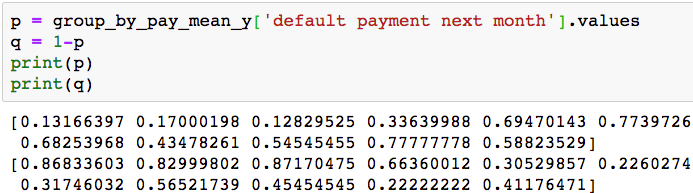


Figure 3.28: Calculating q from p

1. Calculate the odds ratio from p and q, as well as the log odds, using the natural logarithm function from NumPy:

odds\_ratio = p/q

log\_odds = np.log(odds\_ratio)

log\_odds

The output should look like this:

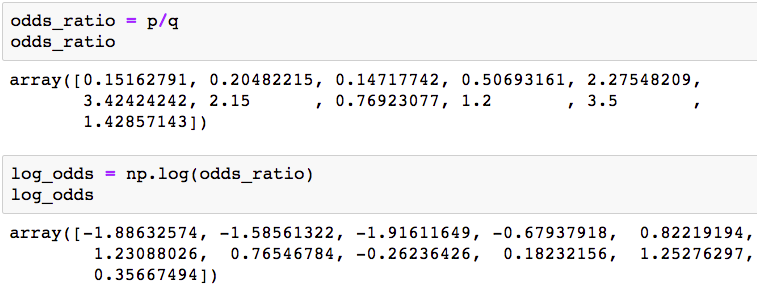


Figure 3.29: Odds ratio and log odds

1. In order to plot the log odds against the values of the feature, we can get the feature values from the index of the DataFrame containing groupby/mean. You can show the index like this:

group\_by\_pay\_mean\_y.index

This should produce the following output:

Int64Index([-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8], dtype='int64', name='PAY\_1')

1. Create a similar plot to what we have already done, to show the log odds against the values of the feature. Here is the code:

plt.plot(group\_by\_pay\_mean\_y.index, log\_odds, '-x')

plt.ylabel('Log odds of default')

plt.xlabel('Values of PAY\_1')

The plot should look like this:

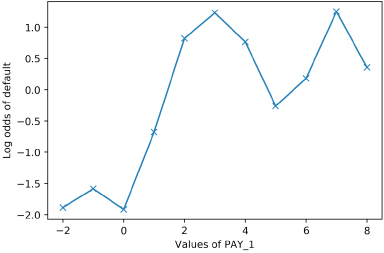


Figure 3.30: Log odds of default for values of PAY\_1

We can see in this plot that the relationship between the log odds of the response variable and the PAY\_1 feature is not all that different from the relationship between the rate of default and this feature that we plotted in Exercise 3.02, Visualizing the Relationship between the Features and Response Variable. For this reason, if the "rate of default" is a simpler concept for you to communicate to the business partner, it may be preferable. However, in terms of understanding the workings of logistic regression, this plot shows exactly what is assumed to be linear.

Is a straight-line fit a good model for this data?

It certainly seems like a "line of best fit" drawn on this plot would go up from left to right. At the same time, this data doesn't seem like it would result in a truly linear process. One way to look at this data is that the values -2, -1, and 0 seem like they lie in a different regime of log odds than the others. PAY\_1 = 1 is sort of intermediate, and the rest are mostly larger. It may be that engineered features based on this variable, or different ways of encoding the categories represented by -2, -1, and 0, would be more effective for modeling. Keep this in mind as we proceed to model this data with logistic regression and then other approaches later in the book.

From Logistic Regression Coefficients to Predictions Using Sigmoid

Before the next exercise, let's take a look at how the coefficients for logistic regression are used to calculate predicted probabilities, and ultimately make predictions for the class of the response variable.

Recall that logistic regression predicts the probability of class membership, according to the sigmoid equation. In the case of two features with an intercept, the equation is as follows:

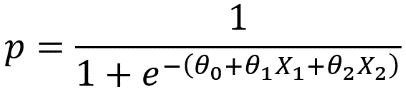


Figure 3.31: Sigmoid function to predict the probability of class membership for two features

When you call the .fit method of a logistic regression model object in scikit-learn using the training data, the 𝜃0, 𝜃1, and 𝜃2 parameters (intercept and coefficients) are estimated from this labeled training data. Effectively, scikit-learn figures out how to choose values for 𝜃0, 𝜃1, and 𝜃2, so that it will classify as many training data points correctly as possible. We'll gain some insight into how this process works in the next chapter.

When you call .predict, scikit-learn calculates predicted probabilities according to the fitted parameter values and the sigmoid equation. A given sample will then be classified as positive if p ≥ 0.5, and negative otherwise.

We know that the plot of the sigmoid equation looks like the following, where the x axis shows the linear combination of the features X = 𝜃0 + 𝜃1X1 + 𝜃2X2:

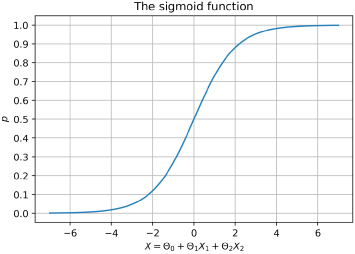


Figure 3.32: Predictions and true classes plotted together

Notice here that if X = 𝜃o + 𝜃1X1 + 𝜃2X2 ≥ 0 on the x axis, then the predicted probability would be p ≥ 0.5 on the y axis and the sample would be classified as positive. Otherwise, p ˂ 0.5 and the sample would be classified as negative. We can use this observation to calculate a linear condition for positive prediction, in terms of the X1 and X2 features, using the coefficients and intercept. Solving the inequality for positive prediction, X = 𝜃o + 𝜃1X1 + 𝜃2X2 ≥ 0, for X2, we can obtain a linear inequality similar to a linear equation in y = mx + b form: X2 ≥ -(𝜃1/𝜃2)X1 - (𝜃o/𝜃2).

This will help to see the linear decision boundary of logistic regression in the X1-X2 feature space in the following exercise.

We have now learned, from a theoretical and mathematical perspective, why logistic regression is considered a linear model. We also examined a single feature and considered whether the assumption of linearity was appropriate. It is also important to understand the assumption of linearity, in terms of how flexible and powerful we can expect logistic regression to be. We explore this in the following exercise.

Exercise 3.05: Linear Decision Boundary of Logistic Regression

In this exercise, we illustrate the concept of a decision boundary for a binary classification problem. We use synthetic data to create a clear example of how the decision boundary of logistic regression looks in comparison to the training samples. We start by generating two features, X1 and X2, at random. Since there are two features, we can say that the data for this problem is two-dimensional. This makes it easy to visualize. The concepts we illustrate here generalize to cases of more than two features, such as the real-world datasets you're likely to see in your work; however, the decision boundary is harder to visualize in higher-dimensional spaces.

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 here: https://github.com/TrainingByPackt/Data-Science-Projects-With-Python-Second-Edition/tree/main/Chapter03/Exercise05.

1. Generate the features using the following code:

from numpy.random import default\_rng

rg = default\_rng(4)

X\_1\_pos = rg.uniform(low=1, high=7, size=(20,1))

print(X\_1\_pos[0:3])

X\_1\_neg = rg.uniform(low=3, high=10, size=(20,1))

print(X\_1\_neg[0:3])

X\_2\_pos = rg.uniform(low=1, high=7, size=(20,1))

print(X\_2\_pos[0:3])

X\_2\_neg = rg.uniform(low=3, high=10, size=(20,1))

print(X\_2\_neg[0:3])

You don't need to worry too much about why we selected the values we did; the plotting we do later should make it clear. Notice, however, that we have assigned the true class at the same time, by defining here which points (X1, X2) will be in the positive and negative classes. The result of this is that we have 20 samples each in the positive and negative classes, for a total of 40 samples, and that we have two features for each sample. We show the first three values of each feature for both the positive and negative classes.

The output should be the following:

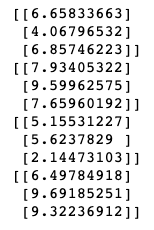


Figure 3.33: Generating synthetic data for a binary classification problem

1. Plot this data, coloring the positive samples as red squares and the negative samples as blue x’s. The plotting code is as follows:

plt.scatter(X\_1\_pos, X\_2\_pos, color='red', marker='s')

plt.scatter(X\_1\_neg, X\_2\_neg, color='blue', marker='x')

plt.xlabel('$X\_1$')

plt.ylabel('$X\_2$')

plt.legend(['Positive class', 'Negative class'])

The result should look like this:

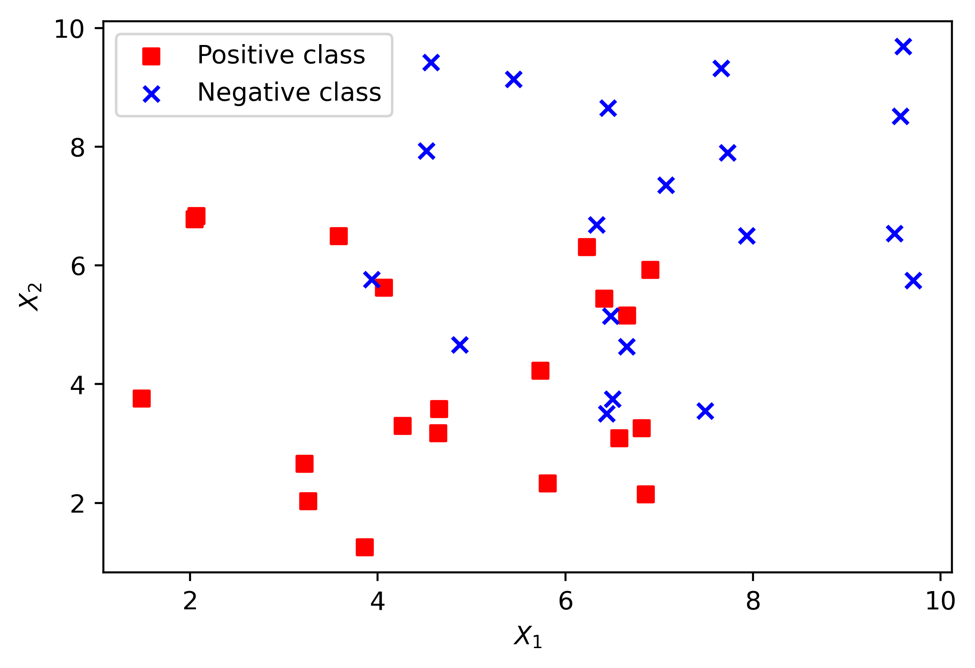


Figure 3.34: Generating synthetic data for a binary classification problem

In order to use our synthetic features with scikit-learn, we need to assemble them into a matrix. We use NumPy's block function for this, to create a 40 by 2 matrix. There will be 40 rows because there are 40 total samples, and 2 columns because there are 2 features. We will arrange things so that the features for the positive samples come in the first 20 rows and those for the negative samples after that.

1. Create a 40 by 2 matrix and then show the shape and the first 3 rows:

X = np.block([[X\_1\_pos, X\_2\_pos], [X\_1\_neg, X\_2\_neg]])

print(X.shape)

print(X[0:3])

The output should be as follows:

(40, 2)

[[6.65833663 5.15531227]

[4.06796532 5.6237829 ]

[6.85746223 2.14473103]]

We also need a response variable to go with these features. We know how we defined them, but we need an array of y values to let scikit-learn know.

1. Create a vertical stack (vstack) of 20 ones and then 20 zeros to match our arrangement of the features and reshape to the way that scikit-learn expects. Here is the code:

y = np.vstack((np.ones((20,1)), np.zeros((20,1)))).reshape(40,)

print(y[0:5])

print(y[-5:])

You will obtain the following output:

[1. 1. 1. 1. 1.]

[0. 0. 0. 0. 0.]

At this point, we are ready to fit a logistic regression model to this data with scikit-learn. We will use all of the data as training data and examine how well a linear model is able to fit the data. The next few steps should be familiar from your work in earlier chapters on how to instantiate a model class and fit the model.

1. First, import the model class using the following code:

from sklearn.linear\_model import LogisticRegression

1. Now instantiate, indicating the liblinear solver, and show the model object using the following code:

example\_lr = LogisticRegression(solver='liblinear')

example\_lr

The output should be as follows:

LogisticRegression(solver='liblinear')

1. Now train the model on the synthetic data:

example\_lr.fit(X, y)

How do the predictions from our fitted model look?

We first need to obtain these predictions, by using the trained model's .predict method on the same samples we used for model training. Then, in order to add these predictions to the plot, we will create two lists of indices to use with the arrays, according to whether the prediction is 1 or 0. See whether you can understand how we've used a list comprehension, including an if statement, to accomplish this.

1. Use this code to get predictions and separate them into indices of positive and negative class predictions. Show the indices of positive class predictions as a check:

y\_pred = example\_lr.predict(X)

positive\_indices = [counter for counter in range(len(y\_pred))

                    if y\_pred[counter]==1]

negative\_indices = [counter for counter in range(len(y\_pred))

                    if y\_pred[counter]==0]

positive\_indices

The output should be as follows:

[2, 3, 4, 5, 6, 7, 9, 11, 13, 15, 16, 17, 18, 19, 26, 34, 36]

From the indices of positive predictions, we can already tell that not every sample in the training data was classified correctly: the positive samples were the first 20 samples, but there are indices outside of that range here. You may have already guessed that a linear decision boundary would not be able to perfectly classify this data, based on examining it. Now let's put these predictions on the plot, in the form of squares and circles around each data point, colored according to positive and negative predictions, respectively: red for positive and blue for negative. You can compare the color and shape of the inner symbols, the true labels of the data, to those of the outer symbols (predictions), to see which points were classified correctly or incorrectly.

1. Here is the plotting code:

plt.scatter(X\_1\_pos, X\_2\_pos, color='red', marker='s')

plt.scatter(X\_1\_neg, X\_2\_neg, color='blue', marker='x')

plt.scatter(X[positive\_indices,0], X[positive\_indices,1],

            s=150, marker='s',

            edgecolors='red', facecolors='none')

plt.scatter(X[negative\_indices,0], X[negative\_indices,1],

            s=150, marker='o',

            edgecolors='blue', facecolors='none')

plt.xlabel('$X\_1$')

plt.ylabel('$X\_2$')

plt.legend(['Positive class', 'Negative class',\

            'Positive predictions', 'Negative predictions'])

The plot should appear as follows:

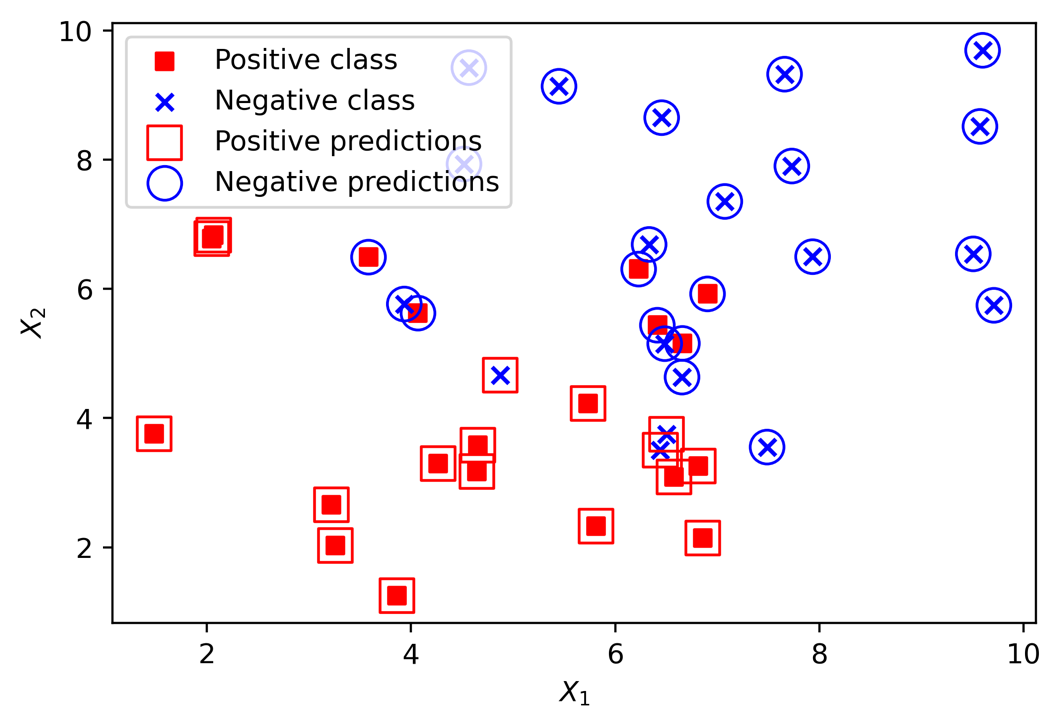


Figure 3.35: Predictions and true classes plotted together

From the plot, it's apparent that the classifier struggles with data points that are close to where you may imagine the linear decision boundary to be; some of these may end up on the wrong side of that boundary. How might we figure out, and visualize, the actual location of the decision boundary? From the previous section, we know we can obtain the decision boundary of a logistic regression, in two-dimensional feature space, using the inequality X2 ≥ -(𝜃1/𝜃2)X1 - (𝜃0/𝜃2). Since we've fitted the model here, we can retrieve the 𝜃1 and 𝜃2 coefficients, as well as the 𝜃0 intercept, to plug into this equation and create the plot.

1. Use this code to get the coefficients from the fitted model and print them:

theta\_1 = example\_lr.coef\_[0][0]

theta\_2 = example\_lr.coef\_[0][1]

print(theta\_1, theta\_2)

The output should look like this:

-0.16472042583006558 -0.25675185949979507

1. Use this code to get the intercept:

theta\_0 = example\_lr.intercept\_

Now use the coefficients and intercept to define the linear decision boundary. This captures the dividing line of the inequality, X2 ≥ -(𝜃1/𝜃2)X1 - (𝜃0/𝜃2):

X\_1\_decision\_boundary = np.array([0, 10])

X\_2\_decision\_boundary = -(theta\_1/theta\_2)\*X\_1\_decision\_boundary

                        - (theta\_0/theta\_2)

To summarize the last few steps, after using the .coef\_ and .intercept\_ methods to retrieve the 𝜃1 and 𝜃2 model coefficients and the 𝜃0 intercept, we then used these to create a line defined by two points, according to the equation we described for the decision boundary.

1. Plot the decision boundary using the following code, with some adjustments to assign the correct labels for the legend, and to move the legend to a location (loc) outside a plot that is getting crowded:

pos\_true = plt.scatter(X\_1\_pos, X\_2\_pos,

                       color='red', marker='s',

                       label='Positive class')

neg\_true = plt.scatter(X\_1\_neg, X\_2\_neg,

                       color='blue', marker='x',

                       label='Negative class')

pos\_pred = plt.scatter(X[positive\_indices,0],

                       X[positive\_indices,1],

                       s=150, marker='s',

                       edgecolors='red', facecolors='none',

                       label='Positive predictions')

neg\_pred = plt.scatter(X[negative\_indices,0],

                       X[negative\_indices,1],

                       s=150, marker='o',

                       edgecolors='blue', facecolors='none',

                       label='Negative predictions')

dec = plt.plot(X\_1\_decision\_boundary, X\_2\_decision\_boundary,

               'k-', label='Decision boundary')

plt.xlabel('$X\_1$')

plt.ylabel('$X\_2$')

plt.legend(loc=[0.25, 1.05])

You will obtain the following plot:

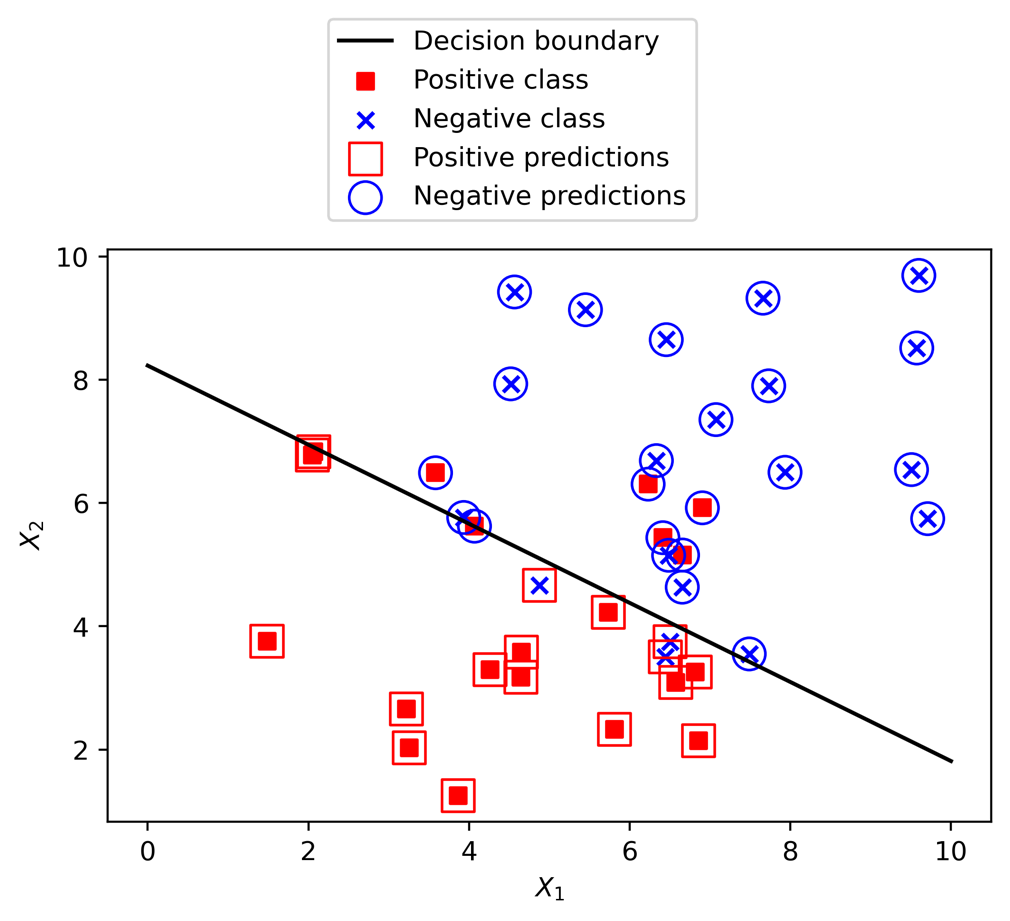


Figure 3.36: True classes, predicted classes, and the decision boundary of a logistic regression

How does the location of the decision boundary compare with where you thought it would be?

Can you see how a linear decision boundary will never perfectly classify this data?

As a way around this, we could create engineered features from existing features here, such as polynomials or interactions, to allow for more complex, non-linear decision boundaries in a logistic regression. Or, we could use non-linear models such as random forest, which can also accomplish this, as we'll see later.

As a final note here, this example was easily visualized in two dimensions since there are only two features. In general, the decision boundary can be described by a hyperplane, which is the generalization of a straight line to multi-dimensional spaces. However, the restrictive nature of the linear decision boundary is still a factor for hyperplanes.

Activity 3.01: Fitting a Logistic Regression Model and Directly Using the Coefficients

In this activity, we're going to train a logistic regression model on the two most important features we discovered in univariate feature exploration, as well as learning how to manually implement logistic regression using coefficients from the fitted model. This will show you how you could use logistic regression in a computing environment where scikit-learn may not be available, but the mathematical functions necessary to compute the sigmoid function are. On successful completion of the activity, you should observe that the calculated ROC AUC values using scikit-learn predictions and those obtained from manual predictions should be the same: approximately 0.63.

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 that can be found here: https://github.com/TrainingByPackt/Data-Science-Projects-With-Python-Second-Edition/tree/main/Chapter03/Activity01.

1. Create a train/test split (80/20) with PAY\_1 and LIMIT\_BAL as features.
2. Import LogisticRegression, with the default options, but set the solver to 'liblinear'.
3. Train on the training data and obtain predicted classes, as well as class probabilities, using the test data.
4. Pull out the coefficients and intercept from the trained model and manually calculate predicted probabilities. You'll need to add a column of ones to your features, to multiply by the intercept.
5. Using a threshold of 0.5, manually calculate predicted classes. Compare this to the class predictions outputted by scikit-learn.
6. Calculate the ROC AUC using both scikit-learn's predicted probabilities and your manually predicted probabilities, and compare them.

Note

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

Summary

In this chapter, we have learned how to explore features one at a time, using univariate feature selection methods including Pearson correlation and an ANOVA F-test. While looking at features in this way does not always tell the whole story, since you are potentially missing out on important interactions between features, it is often a helpful step. Understanding the relationships between the most predictive features and the response variable, and creating effective visualizations around them, is a great way to communicate your findings to your client. We used customized plots, such as overlapping histograms created with Matplotlib, to create visualizations of the most important features.

Then we began an in-depth description of how logistic regression works, exploring such topics as the sigmoid function, log odds, and the linear decision boundary. While logistic regression is one of the simplest classification models, and often is not as powerful as other methods, it is one of the most widely used and is the basis for more sophisticated models such as deep neural networks for classification. So, a detailed understanding of logistic regression can serve you well as you explore more advanced topics in machine learning. And, in some cases, a simple logistic regression may be all that's needed. All other things considered, the simplest model that satisfies the requirements is probably the best model.

If you master the materials in this and the next chapter, you will be well prepared to use logistic regression in your work. In the next chapter, we'll build on the fundamentals we learned here, to see how coefficients are estimated for a logistic regression, as well as how logistic regression can be used effectively with large numbers of features and can also be used for feature selection.

Solution

Activity 3.01: Fitting a Logistic Regression Model and Directly Using the Coefficients

The first few steps are similar to things we've done in previous activities:

1. Create a train/test split (80/20) with PAY\_1 and LIMIT\_BAL as features:

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

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

df[['PAY\_1', 'LIMIT\_BAL']].values,

df['default payment next month'].values,

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

1. Import LogisticRegression, with the default options, but set the solver to 'liblinear':

from sklearn.linear\_model import LogisticRegression

lr\_model = LogisticRegression(solver='liblinear')

1. Train on the training data and obtain predicted classes, as well as class probabilities, using the test data:

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

y\_pred = lr\_model.predict(X\_test)

y\_pred\_proba = lr\_model.predict\_proba(X\_test)

1. Pull out the coefficients and intercept from the trained model and manually calculate predicted probabilities. You'll need to add a column of ones to your features, to multiply by the intercept.

First, let's create the array of features, with a column of ones added, using horizontal stacking:

ones\_and\_features = np.hstack\

                    ([np.ones((X\_test.shape[0],1)), X\_test])

Now we need the intercept and coefficients, which we reshape and concatenate from scikit-learn output:

intercept\_and\_coefs = np.concatenate\

                      ([lr\_model.intercept\_.reshape(1,1), \

                        lr\_model.coef\_], axis=1)

To repeatedly multiply the intercept and coefficients by all the rows of ones\_and\_features, and take the sum of each row (that is, find the linear combination), you could write this all out using multiplication and addition. However, it's much faster to use the dot product:

X\_lin\_comb = np.dot(intercept\_and\_coefs,\

                    np.transpose(ones\_and\_features))

Now X\_lin\_comb has the argument we need to pass to the sigmoid function we defined, in order to calculate predicted probabilities:

y\_pred\_proba\_manual = sigmoid(X\_lin\_comb)

1. Using a threshold of 0.5, manually calculate predicted classes. Compare this to the class predictions outputted by scikit-learn.

The manually predicted probabilities, y\_pred\_proba\_manual, should be the same as y\_pred\_proba; we'll check that momentarily. First, manually predict the classes with the threshold:

y\_pred\_manual = y\_pred\_proba\_manual >= 0.5

This array will have a different shape than y\_pred, but it should contain the same values. We can check whether all the elements of two arrays are equal like this:

np.array\_equal(y\_pred.reshape(1,-1), y\_pred\_manual)

This should return a logical True if the arrays are equal.

1. Calculate ROC AUC using both scikit-learn's predicted probabilities and your manually predicted probabilities, and compare them.

First, import the following:

from sklearn.metrics import roc\_auc\_score

Then, calculate this metric on both versions, taking care to access the correct column, or reshape as necessary:

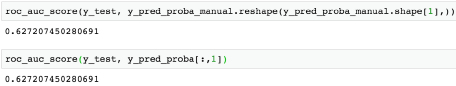


Figure 6.53: Calculating the ROC AUC from predicted probabilities

The AUCs are, in fact, the same. What have we done here? We've confirmed that all we really need from this fitted scikit-learn model is three numbers: the intercept and the two coefficients. Once we have these, we could create model predictions using a few lines of code, with mathematical functions, that are equivalent to the predictions directly made from scikit-learn.

This is good to confirm your understanding, but otherwise, why would you ever want to do this? We'll talk about model deployment in the final chapter. However, depending on your circumstances, you may be in a situation where you don't have access to Python in the environment where new features will need to be input into the model for prediction. For example, you may need to make predictions entirely in SQL. While this is a limitation in general, with logistic regression you can use mathematical functions that are available in SQL to re-create the logistic regression prediction, only needing to copy and paste the intercept and coefficients somewhere in your SQL code. The dot product may not be available, but you can use multiplication and addition to accomplish the same purpose.

Now, what about the results themselves? What we've seen here is that we can slightly boost model performance above our previous efforts: using just LIMIT\_BAL as a feature in the previous chapter's activity, the ROC AUC was a bit less at 0.62, instead of 0.63 here. In the next chapter, we'll learn advanced techniques with logistic regression that we can use to further improve performance.