Overview

Following the completion of exploring the case study data, this chapter introduces the core functionality of scikit-learn for training models and making predictions, through simple use cases of logistic and linear regression. Evaluation metrics for binary classification models, including true and false positive rates, the confusion matrix, the receiver operating characteristic (ROC) curve, and the precision-recall curve, are demonstrated both from scratch and using convenient scikit-learn functionality. By the end of this chapter, you will know the fundamentals of building and evaluating binary classification models using scikit-learn.

Introduction

In the previous chapter, you became familiar with basic Python, and then learned about the pandas tool for data exploration. Using Python and pandas, you performed operations such as loading a dataset, verifying data integrity, and exploratory analyses of the features, or independent variables, in the data.

In this chapter, we finish our exploration of the data by examining the response variable. After we've concluded that the data is of high quality and makes sense, we will be ready to move forward with the practical concerns of developing machine learning models. We will take our first steps with scikit-learn, one of the most popular machine learning packages available in the Python language. Before learning the details of how mathematical models work in the next chapter, here we'll start to get comfortable with the syntax for using them in scikit-learn.

We will also learn some common techniques for answering the question, "Is this model good or not?" There are many possible ways to approach model evaluation. For business applications, a financial analysis to determine the value that could be created by a model is an important way to understand the potential impact of your work. Usually, it’s best to scope the business opportunity of a project at the very beginning. However as the emphasis of this book is on machine learning and predictive modeling, we will demonstrate a financial analysis in the final chapter.

There are several important model evaluation criteria that are considered standard knowledge in data science and machine learning. We will cover a few of the most widely used classification model performance metrics here, to give you a strong foundation.

Exploring the Response Variable and Concluding the Initial Exploration

We have now looked through all the features to see whether any data is missing, as well as to generally examine them. The features are important because they constitute the inputs to our machine learning algorithm. On the other side of the model lies the output, which is a prediction of the response variable. For our problem, this is a binary flag indicating whether or not an account will default the next month, which would have been October for our historical dataset.

The key task for the case study project is to come up with a predictive model for this target. Since the response variable is a yes/no flag, this problem is called a binary classification task. In our labeled data, the samples (accounts) that defaulted (that is, 'default payment next month' = 1) are said to belong to the positive class, while those that didn't belong to the negative class. The main piece of information to examine regarding the response of a binary classification problem is: what is the proportion of the positive class? This is an easy check.

Before we perform this check, we load the packages we need with the following 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

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

Now we load the cleaned version of the case study data like this:

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

Note

The cleaned dataset should have been saved as a result of your work in Chapter 1, Data Exploration and Cleaning. The path to the cleaned data in the preceding code snippet may be different depending on your environment.

Now, to find the proportion of the positive class, all we need to do is get the average of the response variable over the whole dataset. This has the interpretation of the default rate. It's also worthwhile to check the number of samples in each class, using groupby and count in pandas. This is presented in the following screenshot:

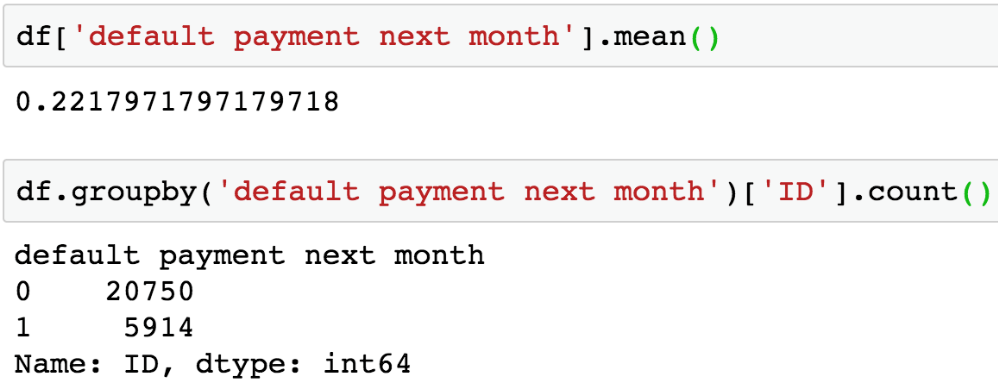


Figure 2.1: Class balance of response variable

Since the target variable is 1 or 0, taking the mean of this column indicates the fraction of accounts that defaulted: 22%. The proportion of samples in the positive class (default = 1), also called the class fraction for this class, is an important statistic. In binary classification, datasets are described in terms of being balanced or imbalanced: are the proportions of positive and negative classes equal, or not? Most machine learning classification models are designed to work with balanced data: a 50/50 split between the classes.

However, in practice, real data is rarely balanced. Consequently, there are several methods geared toward dealing with imbalanced data. These include the following:

* Undersampling the majority class: randomly throwing out samples from the majority class until the class fractions are equal, or at least less imbalanced.
* Oversampling the minority class: randomly adding duplicate samples of the minority class to achieve the same goal.
* Weighting samples: This method is performed as part of the training step, so the minority class collectively has as much "emphasis" as the majority class in the trained model. The effect of this is similar to oversampling.
* More sophisticated methods, such as SMOTE (Synthetic Minority Over-sampling Technique).

While our data is not, strictly speaking, balanced, we also note that a positive class fraction of 22% is not particularly imbalanced, either. Some domains, such as fraud detection, typically deal with much smaller positive class fractions: on the order of 1% or less. This is because the proportion of "bad actors" is quite small compared to the total population of transactions; at the same time, it is important to be able to identify them if possible. For problems like this, it is more likely that using a method to address class imbalance will lead to substantially better results.

Now that we've explored the response variable, we have concluded our initial data exploration. However, data exploration should be considered an ongoing task that you should continually have in mind during any project. As you create models and generate new results, it's always good to think about what those results imply about the data, which usually requires a quick iteration back in to the exploration phase. A particularly helpful kind of exploration, which is also typically done before model building, is examining the relationship between features and the response. We gave a preview of that in Chapter 1, Data Exploration and Cleaning, when we were grouping by the EDUCATION feature and examining the mean of the response variable. We will also do more of this later. However, this has more to do with building a model than checking the inherent quality of the data.

The initial perusal through all the data which we have just completed is an important foundation to lay at the beginning of a project. As you do this, you should ask yourself the following questions:

* Are the data complete?

Are there missing values or other anomalies?

* Are the data consistent?

Does the distribution change over time, and if so, is this expected?

* Do the data make sense?

Do the values of the features fit with their definition in the data dictionary?

The latter two questions help you determine whether you think the data are correct. If the answer to any of these questions is "no," this should be addressed before continuing with the project.

Also, if you think of any alternative, or additional data that might be helpful to have, and possible to get, now would be a good point in the project life cycle to augment your data set with them. Examples of this may include postal-code-level demographic data, which you could join to your data set, if you had the addresses associated with accounts. We don't have these for the case study data and have decided to proceed on this project with the data we have now.

Introduction to Scikit-Learn

While pandas will save you a lot of time in loading, examining, and cleaning data, the machine learning algorithms that will enable you to do predictive modeling are located in other packages. Scikit-learn is a foundational machine learning package for Python, that contains many useful algorithms and has also influenced the design and syntax of other machine learning libraries in Python. For this reason, we focus on scikit-learn to develop skills in the practice of predictive modeling. While it's impossible for any one package to offer "everything," scikit-learn comes pretty close in terms of accommodating a wide range of well-known approaches for classification and regression, and unsupervised learning. That being said, a few other packages you should also be aware of are as follows:

SciPy:

* Most of the packages we've used so far, such as NumPy and pandas, are actually part of the SciPy ecosystem.
* SciPy itself offers lightweight functions for classical approaches such as linear regression and linear programming.

StatsModels:

* More oriented toward statistics and may be more comfortable for users familiar with R
* Can get p-values and confidence intervals on regression coefficients
* Capability for time series models such as ARIMA

XGBoost:

* Offers a suite of state-of-the-art ensemble models that often outperform random forests. We will learn about this package later on.

TensorFlow, Keras, and PyTorch:

* Deep learning capabilities

There are many other Python packages that may come in handy, but this gives you an idea of what’s out there. Now, on to scikit-learn. Scikit-learn offers a wealth of different models for various tasks, but, conveniently, the syntax for using them is consistent. In this section, we will illustrate model syntax using a logistic regression model. Logistic regression, despite its name, is actually a classification model. This is one of the simplest, and therefore most important, classification models. In the next chapter, we will go through the mathematical details of how logistic regression works. Until then, you can simply think of it as a black box that can learn from labeled data, then make predictions.

From the first chapter, you should be familiar with the concept of training an algorithm on labeled data, so that you might use this trained model to then make predictions on new data. Scikit-learn encapsulates these core functionalities in the .fit method for training models, and the .predict method for making predictions. Because of the consistent syntax, you can call .fit and .predict on any scikit-learn model from linear regression to classification trees.

The first step is to choose some model, in this example a logistic regression, and instantiate it from the class provided by scikit-learn. This means you are taking the blueprint of the model that scikit-learn makes available to you and creating a useful object out of it. You can train this object on your data, and then save it to disk for later use. The following snippets can be used to perform this task. The first step is to import the class:

from sklearn.linear\_model import LogisticRegression

The code to instantiate the class in to an object is as follows:

my\_lr = LogisticRegression()

The object is now a variable in our workspace. We can examine it using the following code:

my\_lr

This should give the following output:

LogisticRegression()

Notice that the act of creating the model object involves essentially no knowledge of what logistic regression is or how it works. Although we didn’t select any particular options when creating the logistic regression model object, in fact we are now using many default options for how the model is formulated and would be trained. In effect, these are choices we have made regarding the details of model implementation, without having been aware of it. The danger in an easy-to-use package such as scikit-learn is that it has the potential to obscure these choices from you. However, any time you use a machine learning model that has been prepared for you as scikit-learn models have been, your first job is to understand all the options that are available. A best practice in such cases is to explicitly provide every keyword parameter to the model when you create the object. Even if you are just selecting all the default options, this will help increase your awareness of the choices that are being made.

We will review the interpretation of these choices later on, but for now here is the code for instantiating a logistic regression model with all default options:

my\_new\_lr = LogisticRegression(penalty='l2', dual=False,

tol=0.0001, C=1.0, fit\_intercept=True,

intercept\_scaling=1, class\_weight=None,

random\_state=None, solver='lbfgs',

max\_iter=100, multi\_class='auto',

verbose=0, warm\_start=False,

n\_jobs=None, l1\_ratio=None)

Even though the object we've created here in my\_new\_lr is identical to my\_lr, being explicit like this is especially helpful as you are starting out and learning about different kinds of models. Once you're more comfortable, you may wish to just instantiate with the default options and make changes later as necessary. Here, we show how this may be done. The following code sets two options and displays the current state of the model object:

my\_new\_lr.C = 0.1

my\_new\_lr.solver = 'liblinear'

my\_new\_lr

This should produce the following:

Out[11]:LogisticRegression(C=0.1, solver='liblinear')

Notice that only the options we have updated from the default values are displayed. Here, we've taken what is called a hyperparameter of the model, C, and updated it from its default value of 1 to 0.1. We've also specified a solver. For now, it is enough to understand that hyperparameters are options that you supply to the model, before fitting it to the data. These options specify the way in which the model will be trained. Later, we will explain in detail what all the options are and how you can effectively choose values for them.

To illustrate the core functionality, we will fit this nearly-default logistic regression to some data. As you know by now, supervised learning algorithms rely on labeled data. That means we need both the features, customarily contained in a variable called X, and the corresponding responses, in a variable called y. We will borrow the first 10 samples of one feature, and the response, from our data set to illustrate:

X = df['EDUCATION'][0:10].values.reshape(-1,1)

X

That should show the values of the EDUCATION feature for the first 10 samples:

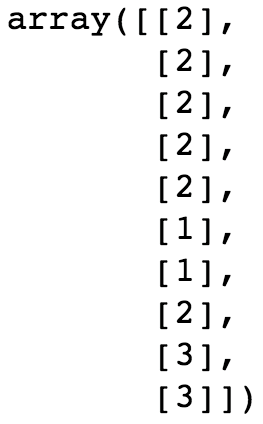


Figure 2.2: First 10 values of a feature

The corresponding first 10 values of the response variable can be obtained as follows:

y = df['default payment next month'][0:10].values

y

Output:

Out[13]: array([1, 1, 0, 0, 0, 0, 0, 0, 0, 0])

Here, we have selected a couple of series (that is, columns) from our DataFrame: the EDUCATION feature we've been discussing, and the response variable. Then we selected the first 10 elements of each, and finally used the .values method to return NumPy arrays. Also notice that we used the .reshape method to reshape the features. Scikit-learn expects that the first dimension (that is, the number of rows) of the array of features will be equal to the number of samples, so we need to make that reshaping for X, but not for y. The –1 in the first positional argument of .reshape means to make the output array shape flexible in that dimension, according to how much data goes in. Since we just have a single feature in this example, we specified the number of columns as the second argument, 1, and let the –1 argument indicate that the array should "fill up" along the first dimension with as many elements as necessary to accommodate the data, in this case 10 elements. Note that while we've extracted the data in to NumPy arrays to show how this can be done, it's also possible to use pandas series as direct input to scikit-learn.

Let's now use this data to fit our logistic regression. This is accomplished with just one line:

my\_new\_lr.fit(X, y)

Output:

Out[14]:LogisticRegression(C=0.1, solver='liblinear')

That's all there is to it! Once the data is prepared and the model is specified, fitting the model almost seems like an afterthought. Of course, we are ignoring all the important options and what they mean right now. But, technically speaking, fitting a model is very easy in terms of the code. You can see that the output of this cell just prints the same options we’ve already seen. While we have not returned anything from the fitting procedure aside from this output, a very important change has taken place. The my\_new\_lr model object is now a trained model. We say that this change happened in place since no new object was created; the existing object, my\_new\_lr, has been modified. This is similar to modifying a DataFrame in place. We can now use our trained model to make predictions on new features, that the model has never "seen" before. Let's try the next 10 rows from the EDUCATION feature.

We can select and view these features using a new variable, new\_X:

new\_X = df['EDUCATION'][10:20].values.reshape(-1,1)

new\_X

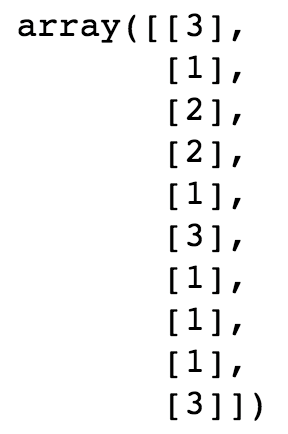


Figure 2.3: New features to make predictions for

Making predictions is done like this:

my\_new\_lr.predict(new\_X)

Output:

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

We can also view the true values corresponding to these predictions, since these data are labeled:

df['default payment next month'][10:20].values

Output:

Out[17]:array([0, 0, 0, 1, 0, 0, 1, 0, 0, 0])

Here, we've illustrated several things. After getting our new feature values, we've called the .predict method on the trained model. Notice that the only argument to this method is a set of features, that is, an "X" that we've called new\_X. There is no "y" supplied in this case. In fact, that is the whole point of predictive modeling: you don't necessarily know the true value of the response variable, so you need to be able to predict it. We see that the output of the .predict method is the model's predictions for those samples whose features are contained in new\_X. We then compare these predictions to the actual true values of the response variable, which we pull from the DataFrame.

How did our little model do? We may naively observe that since the model predicted all 0s, and 80% of the true labels are 0s, we were right 80% of the time, which seems pretty good. On the other hand, we entirely failed to successfully predict any 1s. So, if those were important, we did not actually do that well at all. While this is just a toy example to get you familiar with how scikit-learn works, it's worth considering what a "good" prediction might look like for this problem. We will get into the details of assessing model predictive capability shortly. For now, congratulate yourself on having gotten your hands dirty with some real data, and fitting your first machine learning model.

Generating Synthetic Data

You are now familiar with a variety of Python packages. In fact, these are enough to complete a huge range of data science projects, from classification and regression, to unsupervised learning. And you have seen how fitting a model works in scikit-learn, one of the most useful machine learning packages in Python. In the following exercise, you will walk through the model fitting process on your own.

In order to have data to work with, you will actually generate your own synthetic data. Synthetic data is a valuable learning tool for illustrating mathematical concepts and also for conducting thought experiments to test various ideas. In order to make synthetic data, we will again illustrate here how to use NumPy's random library to generate random numbers, as well as matplotlib's scatter and plot functions to create scatter and line plots. In the exercise, we'll use scikit-learn for the linear regression part. We don't go in to the details of linear regression here, although you are likely familiar with the concept of a line of best fit, which is the same thing.

To get started, we use NumPy to make a one-dimensional array of feature values, X, consisting of 1,000 random real numbers (in other words not just integers but decimals as well) between 0 and 10. We again use a seed for the random number generator. Next, we use numpy.random.uniform, which draws from the uniform distribution: it's equally likely to choose any number between low (inclusive) and high (exclusive), and will return an array of whatever size you specify. We create a one-dimensional array (that is, a vector) with 1,000 elements, then examine the first 10. All of this can be done using the following code:

from numpy.random import default\_rng

rg = default\_rng(12345)

X = rg.uniform(low=0.0, high=10.0, size=(1000,))

X[0:10]

The output should appear as follows:



Figure 2.4: Creating random, uniformly distributed numbers with NumPy

Data for a Linear Regression

Now we need a response variable. For this example, we'll generate data that follows the assumptions of linear regression: the data will exhibit a linear trend but have normally distributed errors. It is unlikely that a real-world data set would satisfy the formal statistical assumptions of a model like this, but you can create a synthetic data set that essentially does. Linear regression theoretically should only be used to model data where the response is a linear transformation of the features, with normally distributed (also called Gaussian) noise:



Figure 2.5: Linear equation with Gaussian noise

Here a is the slope, b is the intercept, and the Gaussian noise has a mean of µ with a standard deviation of σ. In order to write code to implement this, we need to make a corresponding vector of responses, y, which are calculated as a slope times the feature array, X, plus some Gaussian noise (again using NumPy), and an intercept.

To create the linear regression data, we declare variables for the slope and intercept then plug them in to the equation for a line, with the addition of some Gaussian noise: an array of 1,000 data points with the same shape (size) as the feature array, X, where the mean of the noise (loc) is 0 and the standard deviation (scale) is 1. This will add a little "spread" to our linear data. The code to create linear data with Gaussian noise is as follows:

slope = 0.25

intercept = -1.25

y = slope \* X + rg.normal(loc=0.0, scale=1.0, size=(1000,)) + intercept

Now we'd like to visualize these data. We will use matplotlib to plot y against the feature X as a scatter plot. First we use .rcParams to set the resolution (dpi = dots per inch) for a nice crisp image. Then we create the scatter plot with plt.scatter, where the X and y data are the first two arguments, respectively, and the s argument specifies a size for the dots.

This code can be used for plotting:

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

plt.scatter(X,y,s=1)

plt.xlabel('X')

plt.ylabel('y')

After executing these cells, you should see something like this in your notebook:

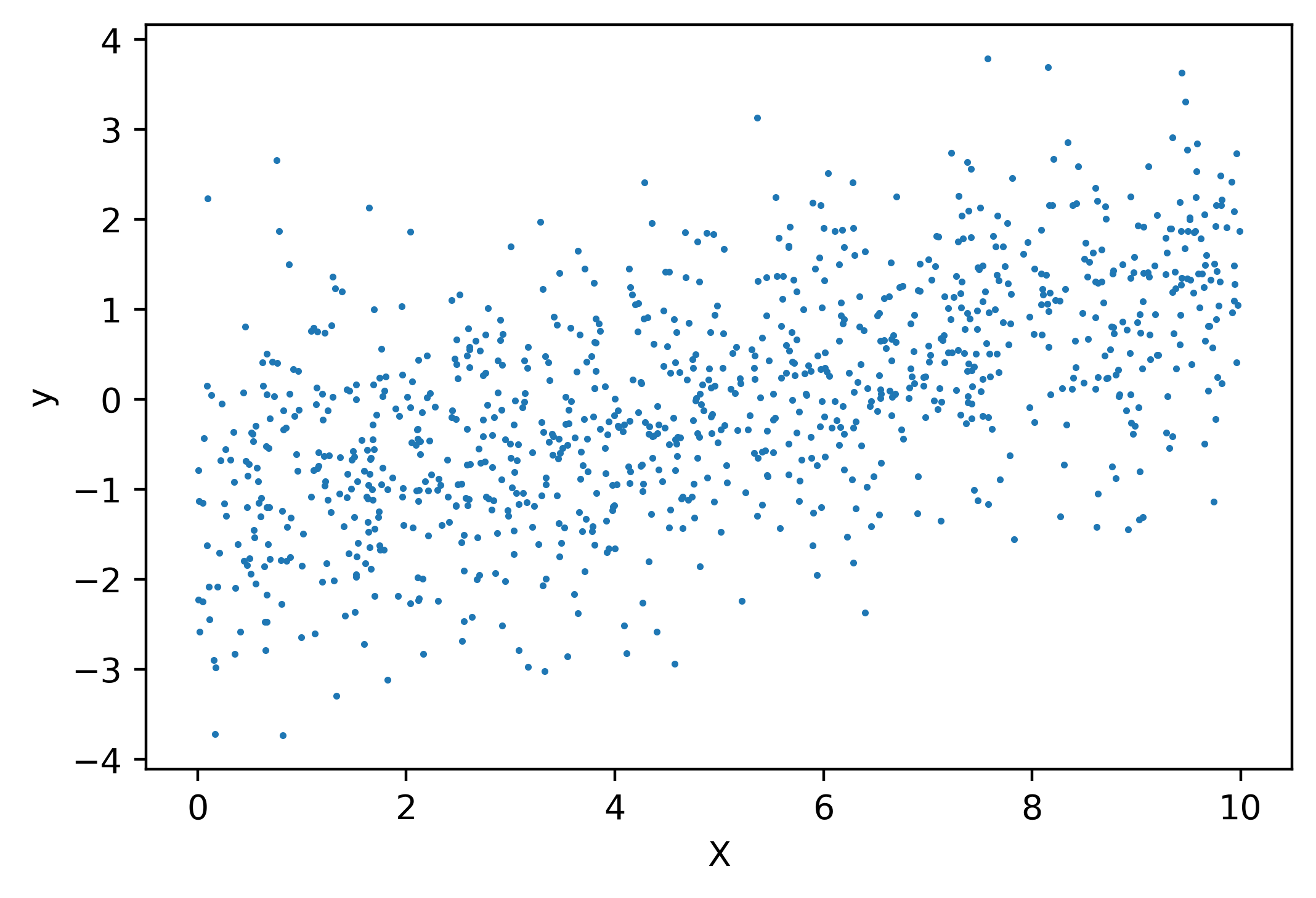


Figure 2.6: Plot the noisy linear relationship

Looks like some noisy linear data, just like we hoped! Now we need to model it.

Exercise 8: Linear Regression in Scikit-Learn

In this exercise, we will take the synthetic data we just generated and determine a line of best fit, or linear regression, using scikit-learn. The first step is to import a linear regression model class from scikit-learn and create an object from it. The import is similar to the LogisticRegression class we imported previously. As with any model class, you should observe what all the default options are. Notice that here for a linear regression, there are not that many options to specify: you will use the defaults for this exercise. The default settings include fit\_intercept=True, meaning the regression model will include an intercept term. This is certainly appropriate since we added an intercept to the synthetic data. Perform the following steps to complete the exercise, noting that the code creating the data for linear regression from the preceding section must be run first in the same notebook (as seen on github):

Note

For Exercise 8, the code and the resulting output have been loaded in a Jupyter Notebook that can be found here: [~~http://bit.ly/2UWPgYo~~](http://bit.ly/2UWPgYo). ~~You can scroll to the appropriate section within the Jupyter Notebook to locate the exercise or activity of choice.~~

1. Execute this code to import the linear regression model class and instantiate it with all default options:

from sklearn.linear\_model import LinearRegression

lin\_reg = LinearRegression(fit\_intercept=True, normalize=False,

copy\_X=True, n\_jobs=None)

lin\_reg

You should see the following output:

Out[11]:LinearRegression()

Where no options are displayed since we used all defaults. Now we can fit the model using our synthetic data, remembering to reshape the feature array, as we did earlier, so that that samples are along the first dimension. After fitting the linear regression model, we examine lin\_reg.intercept\_, which contains the intercept of the fitted model, just as one would think, as well as lin\_reg.coef\_, which contains the slope.

1. Run this code to fit the model and examine the coefficients:

lin\_reg.fit(X.reshape(-1,1), y)

print(lin\_reg.intercept\_)

print(lin\_reg.coef\_)

You should see this output for the intercept and slope:



Figure 2.7: Fitting the regression and examining the parameters

We again see that actually fitting a model in scikit-learn, once the data is prepared and the options for the model are decided, is a trivial process. This is because all the algorithmic work of determining the model parameters is abstracted away from the user. We will discuss this process later, for the logistic regression model we'll use on the case study data.

What about the slope and intercept of our fitted model?

These numbers are fairly close to the slope and intercept we indicated when creating the model. However, because of the random noise, they are only approximations.

Finally, we can use the model to make predictions on feature values. Here, we do this using the same data used to fit the model: the array of features, X. We capture the output of this as a variable, y\_pred. This is very similar to the example shown in Figure 2.8, only here we are making predictions on the same data used to fit the model (previously we made predictions on different data) and here we put the output of the .predict method into a variable.

1. Run this code to make predictions:

y\_pred = lin\_reg.predict(X.reshape(-1,1))

We can plot the predictions, y\_pred, against feature X as a line plot, over the scatter plot of the feature and response data like we made in Figure 2.6. Here we make the addition of plt.plot, which produces a line plot by default, to plot the feature and the model-predicted response values for the model training data. Notice that we follow the X and y data with 'r' in our call to plt.plot. This keyword argument causes the line to have a red color and is part of a shorthand syntax for plot formatting.

1. This code can be used to plot the raw data, as well as the fitted model predictions on these data:

plt.scatter(X,y,s=1)

plt.plot(X,y\_pred,'r')

plt.xlabel('X')

plt.ylabel('y')

After executing this cell, you should see something like this:

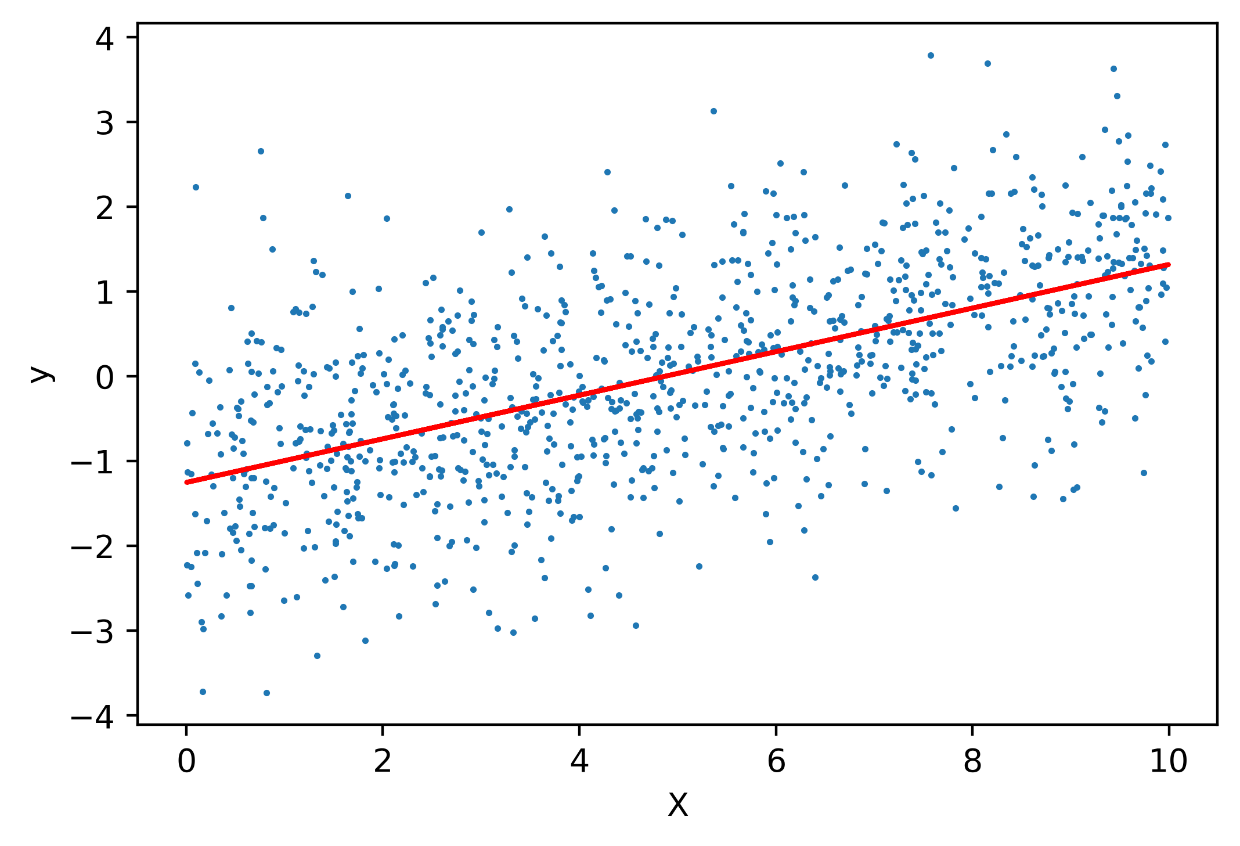


Figure 2.8: Plotting the data and the regression line

The plot shows what the line of best fit to the data looks like, just as we'd expect.

In this exercise, as opposed to when we called .predict with logistic regression, we made predictions on the same data X that we used to train the model. This is an important distinction. While here, we are seeing how the model "fits" the same data that it was trained on, we previously examined model predictions on new, unseen data. In machine learning, we are usually concerned with predictive capabilities: we want models that can help us know the likely outcomes of future scenarios. However, it turns out that model predictions on both the training data used to fit the model, and the testing data, which was not used to fit the model, are important for understanding the workings of the model. We will formalize these notions later in Chapter 4, The Bias-Variance Trade-off when we discuss the bias-variance tradeoff.

Model Performance Metrics for Binary Classification

Before we start building predictive models in earnest, we would like to know how we can determine, once we've created a model, whether it is "good" in some sense of the word. As you may imagine, this question has received a lot of attention from researchers and practitioners. Consequently, there is a wide variety of model performance metrics to choose from.

Note

For an idea of the range of options, have a look at the scikit-learn model evaluation page: https://scikit-learn.org/stable/modules/model\_evaluation.html#model-evaluation.

When selecting a model performance metric to assess the predictive quality of a model, it's important to keep two things in mind.

Appropriateness of the metric for the problem

Metrics are typically only defined for a specific class of problems, such as classification or regression. For a binary classification problem, several metrics characterize the correctness of the yes or no question that the model answers. An additional level of detail here is how often the model is correct for each class, the positive and negative classes. We will go in to detail on these metrics here. On the other hand, regression metrics are aimed at measuring how close a prediction is to the target quantity. If we are trying to predict the price of a house, how close did we come? Are we systematically over- or under-estimating? Are we getting the more expensive houses wrong but the cheaper ones right? There are many possible ways to look at regression metrics.

Does the metric answer the business question?

Whatever class of problem you are working on, there will be many choices for the metric. Which one is the right one? And even then, how do you know if a model is "good enough" in terms of the metric? At some level, this is a subjective question. However, we can be objective when we consider what the goal of the model is. In a business context, typical goals are to create profit or reduce loss. Ultimately, you need to unify your business question, which is often related to money in some way, and the metric you will use to judge your model. For example, in our credit default problem, is there a particularly high cost associated with not correctly identifying accounts that will default? Is this more important than potentially misclassifying some of the accounts that won't default?

Later in the book, we'll incorporate the concept of relative costs and benefits of correct and incorrect classifications in our problem and conduct a financial analysis. First, we'll introduce you to the most common metrics used to assess predictive quality of binary classification models, the kind of models we need to build for our case study.

Splitting the Data: Training and Testing sets

In the scikit-learn introduction of this chapter, we introduced the concept of using a trained model to make predictions on new data that the model had never "seen" before. It turns out this is a foundational concept in predictive modeling. In our quest to create a model that has predictive capability, we need some kind of measure of how well the model can make predictions on data that were not used to fit the model. This is because in fitting a model, the model becomes "specialized" at learning the relationship between features and response, on the specific set of labeled data that were used for fitting. While this is nice, in the end we want to be able to use the model to make accurate predictions on new, unseen data, for which we don't know the true value of the labels.

For example, in our case study, once we deliver the trained model to our client, they will then generate a new data set of features like those we have now, except instead of spanning the period from April to September, they will span from May to October. And our client will be using the model with these features, to predict whether accounts will default in November.

In order to have knowledge of how well we can expect our model to predict which accounts will actually default in November (which won't be known until December), we can take our current data set, and reserve some of the data we have, with known labels, from the model training process. This data is referred to as testing data and may also be called out-of-sample data since it consists of samples that were not used in training the model. Those samples used to train the model are called training data. The practice of holding out a set of testing data gives us an idea of how the model will perform when it is used for its intended purpose, to make predictions on samples that were not included during model training. In this chapter we'll create an example train/test split to illustrate different binary classification metrics.

We will use the convenient train\_test\_split functionality of scikit-learn to split the data so that 80% will be used for training, holding 20% back for testing. These percentages are a common way to make such a split; in general, you want enough training data to allow the algorithm to adequately "learn" from a representative sample of data. However, these percentages are not set in stone. If you have a very large number of samples, you may not need as large a percentage of training data, since you will be able to achieve a pretty large, representative training set with a lower percentage. We encourage you to experiment with different sizes and see the effect. Also, be aware that every problem is different, with respect to how much data is needed to effectively train a model. There is no hard and fast rule for sizing your training and testing sets.

For our 80/20 split, consider the code shown in the following snippet:

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

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

df['EDUCATION'].values.reshape(-1,1),

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

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

Notice in the preceding snippet that we've indicated the test\_size as 0.2, or 20%. The size of the training data will be automatically set to the remainder, 80%. Let's examine the shapes of our training and testing data, to see whether they are as expected as shown in the following output:

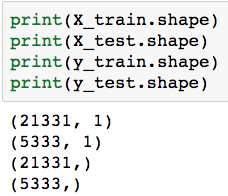


Figure 2.9: Shape of training and testing sets

You should confirm for yourself that the number of samples (rows) in the training and testing sets is consistent with an 80/20 split.

In making the train/test split, we've also set the random\_state parameter, which is a random number seed. Using this parameter allows for a consistent train/test split across runs of this notebook. Otherwise, the random splitting procedure would select a different 20% of the data for testing each time the code was run. Finally, observe that similar to our initial illustration of model fitting, we are still only going to use a single feature here: EDUCATION. We are doing this just to show how to split data and calculate different model performance metrics. We will try other features later when we go in to detail on how different machine learning models work, and seek to create the best model we can to deliver to our client.

The first argument to train\_test\_split is the features, in this case just EDUCATION, and the second argument is the response. There are four outputs: the features of the samples in the training and testing sets, respectively, and the corresponding response variables that go with these sets of features. All this function has done is randomly select 20% of the row indices from the data set and subset out these features and responses as testing data, leaving the rest for training. Now that we have our training and testing data, it's good to make sure the nature of the data is the same between these sets. In particular, is the fraction of the positive class similar? You can observe this in the following output:

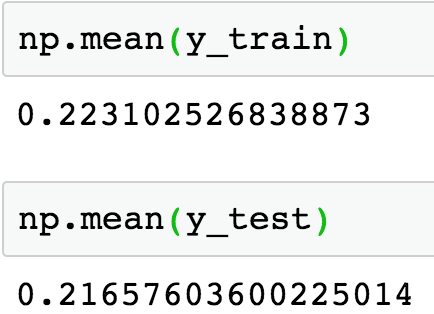


Figure 2.10: Class fractions in training and testing data

The positive class fractions in the training and testing data are both about 22%. This is good, as these are the same as the overall data, and we can say that the training set is representative of the testing set. In this case, since we have a pretty large data set with tens of thousands of samples, and the classes are not too imbalanced, we don't have to take precautions to ensure this happens.

However, you can imagine that if the data set is smaller, and the positive class is very rare, it may happen that the class fractions are noticeably different between the training and testing sets, or worse yet, there are no positive samples at all in the testing set! In order to guard against such scenarios, you could use stratified sampling, with the stratify keyword argument of train\_test\_split. This procedure also makes a random split of the data in to training and testing sets but guarantees that the class fractions will be equal or very similar between the two.

Note

Out-of-time testing

If your data contains both features and responses that span a substantial period of time, it's a good practice to try making your train/test split over time. For example, if you have two years of data with features and responses from every month, you may wish to try sequentially training the model on 12 months of data and testing on the next month, or the month after that, depending on what is operationally feasible when the model will be used. You could repeat this until you've exhausted your data, to get a few different testing scores. This will give you useful insight in to model performance, because it simulates the actual conditions the model will face when it is deployed: a model trained on older features and responses will be used to make predictions on newer data. In the case study, the responses only come from one point in time (credit defaults within one month), so this is not an option here.

Classification Accuracy

Now we proceed to fit an example model, to illustrate binary classification metrics. We will continue to use a logistic regression with near-default options, choosing the same options we demonstrated in Chapter 1, Data Exploration and Cleaning:

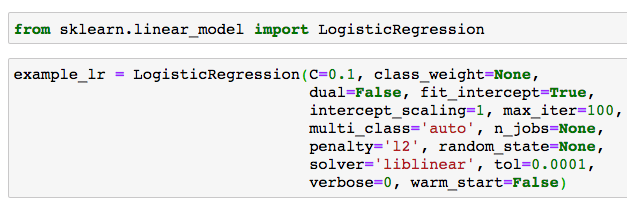


Figure 2.11: Loading the model class and creating a model object

Now we proceed to train the model, as you might imagine, using the labeled data from our training set. We proceed immediately to use the trained model to make predictions on the features of the samples from the held-out test set:

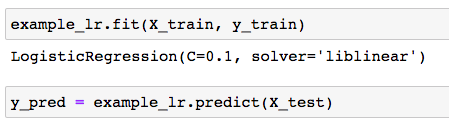


Figure 2.12: Training a model and making predictions on the test set

We've stored the model-predicted labels of the test set in a variable called y\_pred. How should we now assess the quality of these predictions? We have the true labels, in the y\_test variable. First, we will compute what is probably the simplest of all binary classification metrics: accuracy. Accuracy is defined as the proportion of samples that were correctly classified.

One way to calculate accuracy is to create a logical mask that is True whenever the predicted label is equal to the actual label, and False otherwise. We can then take the average of this mask, which will interpret True as 1 and False as 0, giving us the proportion of correct classifications:

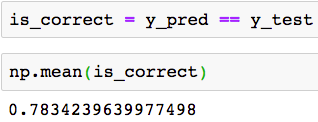


Figure 2.13: Calculating classification accuracy with a logical mask

This indicates the model is correct 78% of the time. While this is a pretty straightforward calculation, there are actually easier ways to calculate accuracy using the convenience of scikit-learn. One way is to use the trained model's .score method, passing the features of the testing data to make predictions on, and the testing labels. This method makes the predictions and then does the same calculation we performed previously, all in one step. Or, we could import scikit-learn's metrics library, which includes many model performance metrics such as accuracy\_score. For this, we pass the true labels and the predicted labels:

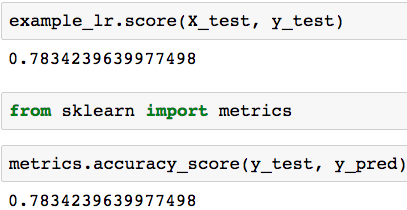


Figure 2.14: Calculating classification accuracy with scikit-learn

These all give the same result, as they should. Now that we know how accurate the model is, how do we interpret this metric? On the surface, an accuracy of 78% may sound good. We are getting most of the predictions right. However, an important test for the accuracy of binary classification is to compare things to a very simple hypothetical model that only makes one prediction: this hypothetical model predicts the majority class for every sample, no matter what the features are. While in practice this model is useless, it provides an important extreme case with which to compare the accuracy of our trained model. Such extreme cases are sometimes referred to as null models.

Think about what the accuracy of such a null model would be. In our data set, we know that about 22% of the samples are positive. So, the negative class is the majority class, with the remaining 78% of the samples. Therefore, a null model for this data set, that always predicts the majority negative class, will be right 78% of the time. Now when we compare our trained model here to such a null model, it becomes clear that an accuracy of 78% is actually not very useful. We can get the same accuracy with a model that doesn't pay any attention to the features!

While we can interpret accuracy in terms of a majority-class null model, there are other binary classification metrics that delve a little deeper in to how the model is performing for negative, versus positive samples.

True Positive Rate, False Positive Rate, and Confusion Matrix

In binary classification, there are just two labels to consider: positive and negative. As a more descriptive way to look at model performance than the accuracy of prediction across all samples, we can also look at the accuracy of only those samples that have a positive label. The proportion of these that we successfully predict as positive, is called the true positive rate (TPR). If we say that P is the number of samples in the positive class in the testing data, and TP is the number of true positives, defined as the number of positive samples that were predicted to be positive by the model, then the TPR is as follows:

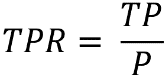


Figure 2.15: True positive rate equation

The flip side of the true positive rate is the false negative rate (FNR). This is the proportion of positive testing samples that we incorrectly predicted as negative. Such errors are called false negatives (FN) and the false negative rate (FNR) is calculated as follows:

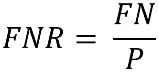


Figure 2.16: False negative rate equation

Since all the positive samples are either correctly or incorrectly predicted, the sum of the number of true positives and the number of false negatives equals the total number of positive samples. Mathematically, P = TP + FN and therefore, using the definitions of TPR and FNR, we have the following:



Figure 2.17: The relation between TPR and FNR

Since the TPR and FNR sum to 1, it's sufficient to just calculate one of them.

Similar to the TPR and FNR, there is the true negative rate (TNR) and the false positive rate (FPR). If N is the number of negative samples, the sum of true negative samples (TN) is the number of these that are correctly predicted, and the sum of false positive (FP) samples is the number incorrectly predicted as positive:

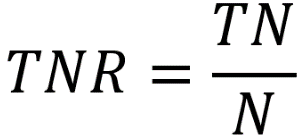


Figure 2.18: True negative rate equation

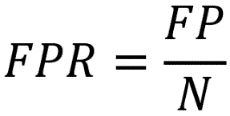


Figure 2.19: False positive rate equation



Figure 2.20: Relation between true negative rate and false positive rate

True and false positives and negatives can be conveniently summarized in a table called a confusion matrix. A confusion matrix for a binary classification problem is a 2 x 2 matrix where the true class is along one axis and the predicted class is along the other. The confusion matrix gives a quick summary of how many true and false positives and negatives there are:

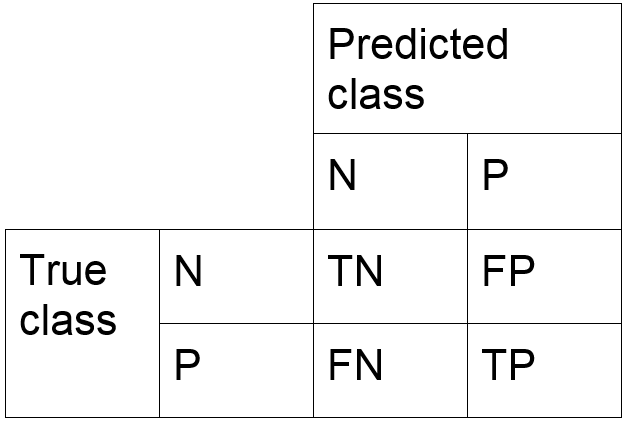


Figure 2.21: The confusion matrix for binary classification

Since we hope to make correct classifications, we hope that the diagonal (that is, the entries along a diagonal line from the top left to the bottom right: TN and TP) of the confusion matrix are relatively large, while the off-diagonals are relatively small, as these represent incorrect classifications.

Exercise 9: Calculating the True and False Positive and Negative Rates and Confusion Matrix in Python

In this exercise, we'll use the testing data and model predictions from the logistic regression model we created previously, using only the EDUCATION feature. We will illustrate how to manually calculate the true and false positive and negative rates, as well as the numbers of true and false positives and negatives needed for the confusion matrix. Then we show a quick way to calculate a confusion matrix with scikit-learn. Perform the following steps to complete the exercise, noting that some code from the previous section must be run before doing this exercise (as seen on github):

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can found here: [~~http://bit.ly/2UWPgYo~~](http://bit.ly/2UWPgYo).

1. Run this code to calculate the number of positive samples:

P = sum(y\_test)

P

The output should appear like this:

1155

Now we need the number of true positives. These are samples where the true label is 1 and the prediction is also 1. We can identify these with a logical mask for the samples that are positive (y\_test==1) AND (& is the logical AND operator in Python) have a positive prediction (y\_pred==1).

1. Use this code to calculate the number of true positives:

TP = sum( (y\_test==1) & (y\_pred==1) )

TP

Output:

0

The true positive rate is the proportion of true positives to positives.

1. Run the following code to obtain the true positive rate:

TPR = TP/P

TPR

You will obtain the following output:

0.0

Similarly, we can identify the false negatives.

1. Calculate the number of false negatives with this code:

FN = sum( (y\_test==1) & (y\_pred==0) )

FN

This should output the following: 1155

We'd also like the false negative rate.

1. Calculate the false negative rate with this code:

FNR = FN/P

FNR

This should output the following:

1.0

What have we learned from the true positive and false negative rates?

First, we can confirm that they sum to 1. This fact is easy to see because the TPR = 0 and the FPR = 1. What does this tell us about our model? On the testing set, at least for the positive samples, the model has in fact acted as a majority class null model. Every positive sample was predicted to be negative, so none of them were correctly predicted.

1. Let's find the TNR and FPR of our testing data. Since these calculations are very similar to those we looked at previously, we show them all at once and illustrate a new Python function:

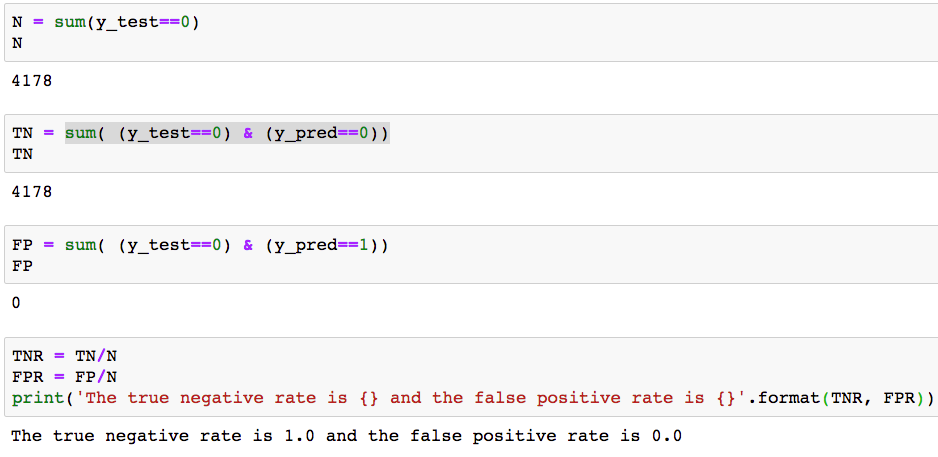


Figure 2.22: Calculating true negative and false positive rates and printing them

In addition to calculating the TNR and FPR in a similar way that we had previously with the TPR and FNR, we demonstrate the print function in Python along with the .format method for strings, which allows substitution of variables in locations marked by curly braces {}. There are a range of options for formatting numbers, such as including a certain number of decimal places.

For additional details, refer to <https://docs.python.org/3/tutorial/inputoutput.html>.

Now, what have we learned here? In fact, our model behaves exactly as the majority class null model for all samples, both positive and negative. It's clear we're going to need a better model.

While we have manually calculated all the entries of the confusion matrix in this exercise, in scikit-learn there is a quick way to do this. Note that in scikit-learn, the true class is along the vertical axis and the predicted class is along the horizontal axis of the confusion matrix, as we presented it earlier.

1. Create a confusion matrix in scikit-learn with this code:

metrics.confusion\_matrix(y\_test, y\_pred)

You will obtain the following output:

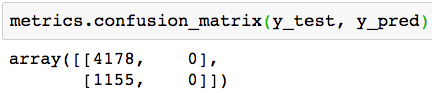


Figure 2.23: The confusion matrix for our example model

All the information we need to calculate the TPR, FNR, TNR, and FPR is contained in the confusion matrix. We also note that there are many more classification metrics that can be derived from the confusion matrix. In fact, some of these are actually synonyms for ones we've already examined here. For example, the TPR is also called recall and sensitivity. Along with recall, another metric that is often used for binary classification is precision: this is the proportion of positive predictions that are correct (as opposed to the proportion of positive samples that are correctly predicted). We'll get more experience with precision in the activity for this chapter.

Note

Multiclass classification

Our case study involves a binary classification problem, with only two possible outcomes: the account does or does not default. Another important type of machine learning classification problem is multiclass classification. In multiclass classification, there are several possible mutually exclusive outcomes. A classic example is image recognition of handwritten digits; a handwritten digit should be only one of 0, 1, 2, … 9. Although multiclass classification is outside the scope of this book, the metrics we are learning now for binary classification can be extended to the multiclass setting.

Discovering Predicted Probabilities: How Does Logistic Regression Make Predictions?

Now that we're familiar with accuracy, true and false positives and negatives, and the confusion matrix, it's time to learn a little bit more about logistic regression, in order to further our knowledge of binary classification metrics. So far, we've only considered logistic regression as a "black box" that can learn from labeled training data and then make binary predictions on new features. While we will learn about the workings of logistic regression in detail later in the book, we can begin to peek inside the black box now.

One thing to understand about how logistic regression works is that the raw predictions – in other words, the direct outputs from the mathematical equation that defines logistic regression – are not binary labels. They are actually probabilities on a scale from 0 to 1 (although, technically, the equation never allows for the probabilities to be exactly equal to 0 or 1, as we'll see later). These probabilities are only transformed into binary predictions through the use of a threshold. The threshold is the probability above which a prediction is declared to be positive, and below which it is negative. The threshold in scikit-learn is 0.5. This means any sample with a predicted probability of at least 0.5 is identified as positive, and any with a predicted probability < 0.5 is decided to be negative. However, we are free to use any threshold we want. In fact, choosing the threshold is one of the key flexibilities of logistic regression, as well as other machine learning classification algorithms that estimate probabilities of class membership.

Exercise 10: Obtaining Predicted Probabilities from a Trained Logistic Regression Model

In the following exercise, we will get familiar with the predicted probabilities of logistic regression and how to obtain them from a scikit-learn model.

We can begin to discover predicted probabilities by further examining the methods available to us on the logistic regression model object that we trained already in this chapter. Recall that before, once we trained the model, we could then make binary predictions using the values of features from new samples, by passing these values to the .predict method of the trained model. These are predictions made on the assumption of a threshold of 0.5.

However, we can directly access the predicted probabilities of these samples, using the .predict\_proba method. Perform the following steps to complete the exercise, keeping in mind that you will need to recreate the same model trained previously in the chapter if you are starting a new notebook.

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can found here: [~~http://bit.ly/2UWPgYo~~](http://bit.ly/2UWPgYo)~~,~~ where the prerequisite steps of training the model are completed prior to the first step shown here.

1. Obtain predicted probabilities for the testing samples using this code:

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

y\_pred\_proba

The output should be as follows:

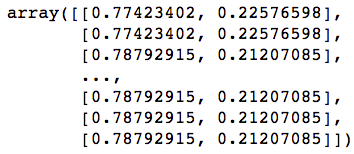


Figure 2.24: Predicted probabilities of the testing data

We see in the output of this, which we've stored in y\_pred\_proba, that there are two columns. This is because there are two classes in our classification problem: negative and positive. Assuming the negative labels are coded as 0 and the positives as 1, as they are in our data, scikit-learn will report the probability of negative class membership as the first column, and positive class membership as the second.

Since the two classes are mutually exclusive, and are the only options, the sum of predicted probabilities for the two classes should equal 1 for every sample. Let's confirm this.

First, we can use np.sum over the first dimension (columns) to calculate the sum of probabilities for each sample.

1. Calculate the sum of predicted probabilities for each sample with this code:

prob\_sum = np.sum(y\_pred\_proba,1)

prob\_sum

The output is as follows:

array([1., 1., 1., ..., 1., 1., 1.])

It certainly looks like all 1s! We should check to see that the result is the same shape as the array of testing data labels.

1. Check the array shape with this code:

prob\_sum.shape

This should output the following:

(5333,)

Good; this is the expected shape. Now, to check that each and every value is 1. We use np.unique to show all the unique elements of this array. This is similar to DISTINCT in SQL. If all the probability sums are indeed 1, there should only be 1 unique element of the probability array: 1.

1. Show all unique array elements with this code:

np.unique(prob\_sum)

This should output the following:

array([1.])

After confirming our belief in the predicted probabilities, we note that since class probabilities sum to 1, it's sufficient to just consider the second column, the predicted probability of positive class membership. Let's capture these in an array.

1. Run this code to put the second column of the predicted probabilities array (predicted probability of membership in positive class) in an array:

pos\_proba = y\_pred\_proba[:,1]

pos\_proba

The output should be as follows:

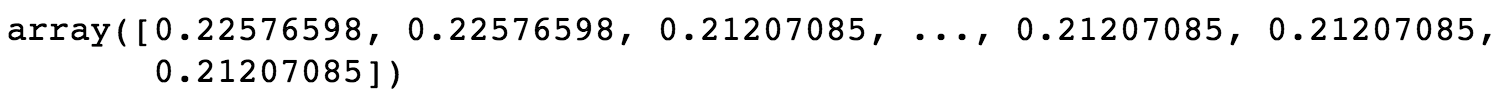


Figure 2.25: Predicted probabilities of positive class membership

What do these probabilities look like? One way to find out, and a good diagnostic for model output, is to plot the predicted probabilities. A histogram is a natural way to do this. We can directly use the matplotlib histogram function, hist(), to do this. Note that if you execute a cell with only the histogram function, you will get the output of the NumPy histogram function returned before the plot. This includes the number of samples in each bin, and the locations of the bin edges.

1. Execute this code to see histogram output and an unformatted plot (not shown):

plt.hist(pos\_proba)

The output is as follows:

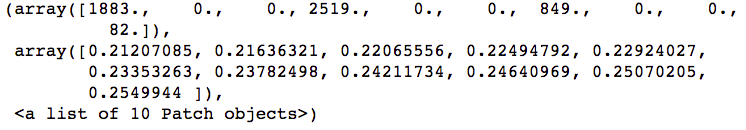


Figure 2.26: Details of histogram calculation

This may be useful information for you, and could also be obtained directly from the np.histogram() function. However, here we're mainly interested in the plot, so we adjust the font size and add some axis labels.

1. Run this code for a formatted histogram plot of predicted probabilities:

mpl.rcParams['font.size'] = 12

plt.hist(pos\_proba)

plt.xlabel('Predicted probability of positive class for testing data')

plt.ylabel('Number of samples')

The plot should look like this:

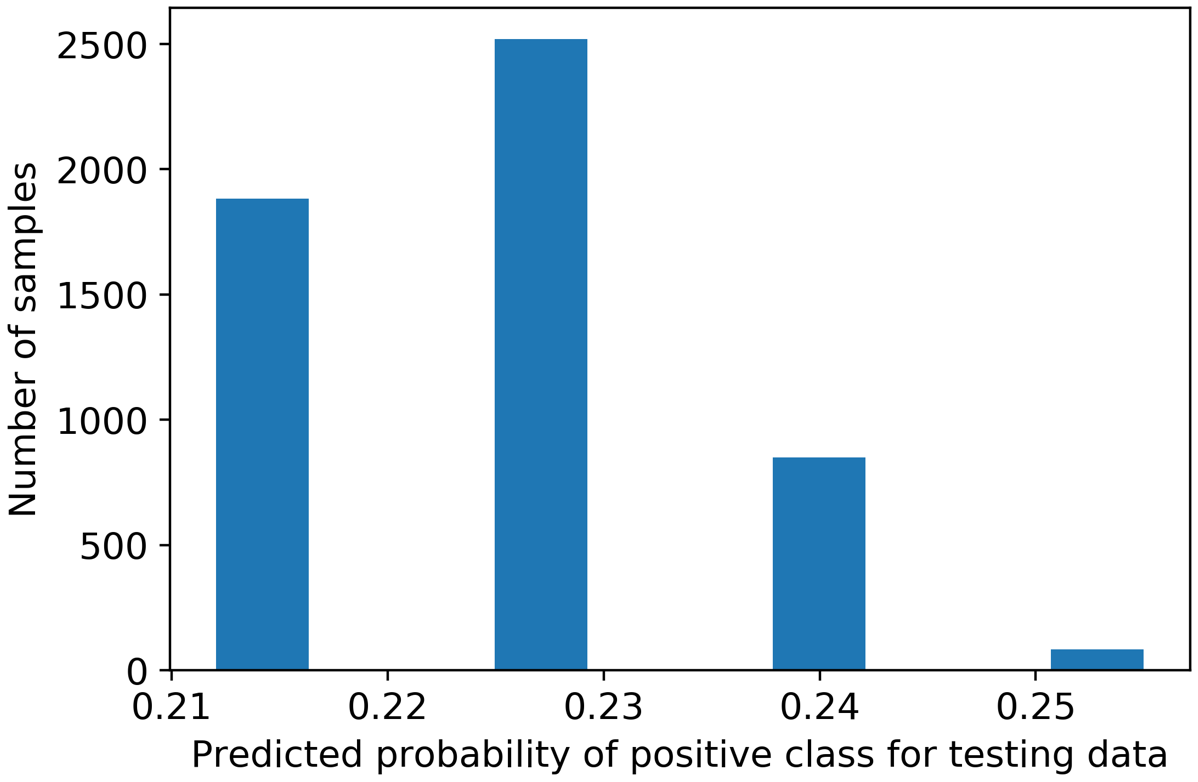


Figure 2.27: Histogram plot of predicted probabilities

Notice that in the histogram of probabilities, there are only four bins that actually have samples in them, and they are spaced fairly far apart. This is because there are only four unique values for the EDUCATION feature, which is the only feature in our example model.

Also, notice that all the predicted probabilities are below 0.5. This is the reason every sample was predicted to be negative, using the 0.5 threshold. Now we can imagine that if we set our threshold below 0.5, we would get different results. For example, if we set the threshold at 0.25, all of the samples in the smallest bin to the far right of Figure 2.27 would be classified as positive, since the predicted probability for all of these is above 0.25. It would be informative for us if we could visually see how many of these samples actually had positive labels. Then we could see whether moving our threshold down to 0.25 would improve the performance of our classifier by classifying the samples in the rightmost bin as positive.

In fact, we can visualize this easily, using a stacked histogram. This will look a lot like the histogram in Figure 2.27, except that the negative and positive samples will be colored differently. First, we need to distinguish between positive and negative samples in the predicted probabilities. We can do this by indexing our array of predicted probabilities with logical masks; first to get positive samples, where y\_test == 1, and then to get negative samples, where y\_test == 0.

1. Isolate the predicted probabilities for positive and negative samples with this code:

pos\_sample\_pos\_proba = pos\_proba[y\_test==1]

neg\_sample\_pos\_proba = pos\_proba[y\_test==0]

Now we want to plot these as a stacked histogram. The code is similar to the histogram we already created, except that we will pass a list of arrays to be plotted, which are the arrays of probabilities for positive and negative samples we just created, and a keyword indicating we'd like the bars to be stacked, as opposed to plotted side by side. We'll also create a legend so that the colors are clearly identifiable on the plot.

1. Plot a stacked histogram using this code:

plt.hist([pos\_sample\_pos\_proba, neg\_sample\_pos\_proba],

histtype='barstacked')

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

plt.xlabel('Predicted probability of positive class')

plt.ylabel('Number of samples')

The plot should look like this:

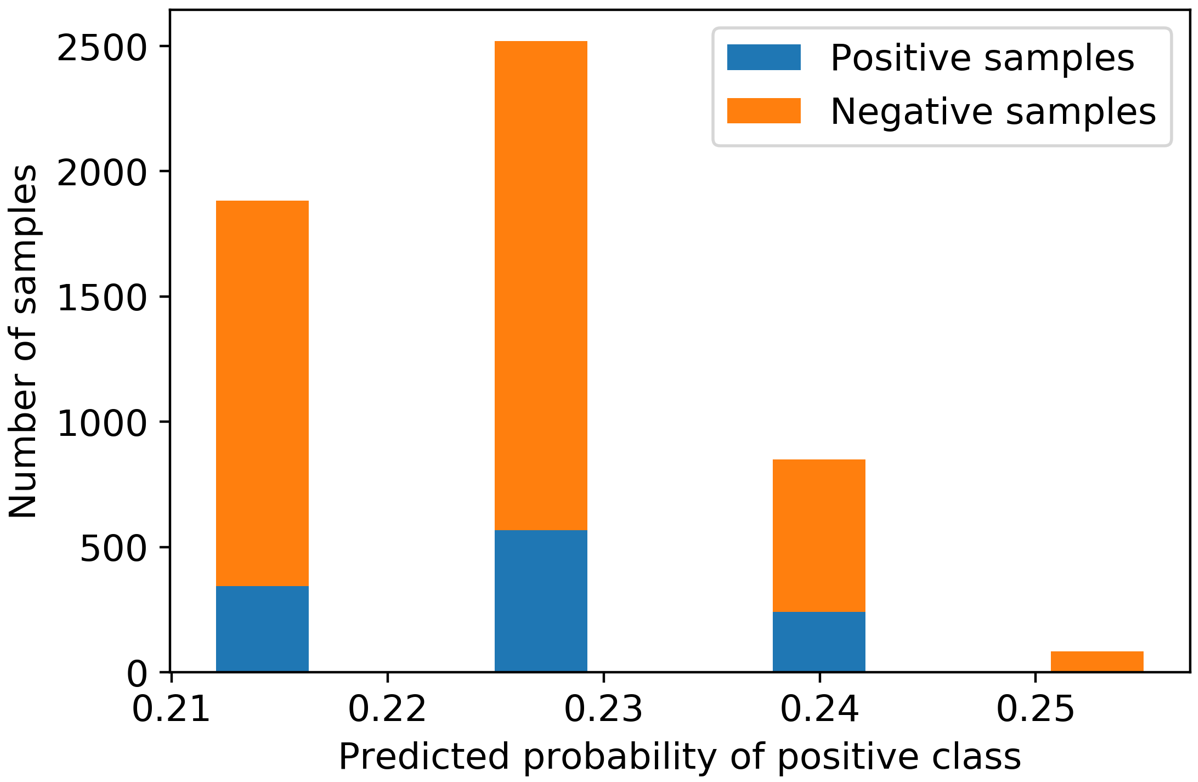


Figure 2.28: Stacked histogram of predicted probabilities by class

The plot shows us the true label of the samples for each predicted probability. Now we can consider what the effect would be of lowering the threshold to 0.25. Take a moment and think about what this would mean, keeping in mind that any sample with a predicted probability at or above the threshold would be classified as positive.

Since nearly all the samples in the small bin to the right of Figure 2.28 are negative samples, if we were to decrease the threshold to 0.25, we would erroneously classify these as positive samples and increase our false positive rate. At the same time, we still wouldn't have managed to classify many, if any, positive samples correctly, so our true positive rate wouldn't increase very much at all. Making this change would appear to decrease the accuracy of the model.

The Receiver Operating Characteristic (ROC) Curve

Deciding on a threshold for a classifier is a question of finding the "sweet spot" where we are successfully recovering enough true positives, without incurring too many false positives. As the threshold is lowered more and more, there will be more of both. A good classifier will be able to capture more true positives without the expense of a large number of false positives. What would be the effect of lowering the threshold even more, with the predicted probabilities from the previous exercise? It turns out there is a classic method of visualization in machine learning, with a corresponding metric, that can help answer these kinds of questions.

The receiver operating characteristic (ROC) curve is a plot of the pairs of true positive rates (y-axis) and false positive rates (x-axis) that result from lowering the threshold down from 1, all the way to 0. You can imagine that if the threshold is 1, there are no positive predictions since a logistic regression only predicts probabilities strictly between 0 and 1 (endpoints not included). Since there are no positive predictions, the TPR and the FPR are both 0, so the ROC curve starts out at (0, 0). As the threshold is lowered, the TPR will start to increase, hopefully faster than the FPR if it's a good classifier. Eventually, when the threshold is lowered all the way to 0, every sample is predicted to be positive, including all the ones that are, in fact, positive, but also all the ones that are actually negative. This means the TPR is 1 but the FPR is also 1. In between these two extremes are the reasonable options for where you may want to set the threshold, depending on the relative costs and benefits of true and false positives and negatives for the specific problem being considered. In this way, it is possible to get a complete picture of the performance of the classifier at all different thresholds to decide which one to use.

We could write the code to determine the TPRs and FPRs of the ROC curve by using the predicted probabilities and varying the threshold from 1 to 0. Instead, we will use scikit-learn's convenient functionality, which will take the true labels and predicted probabilities as inputs, and return arrays of TPRs, FPRs, and the thresholds that lead to them. We will then plot the TPRs against the FPRs to show the ROC curve. Run this code to use scikit-learn to generate the arrays of TPRs and FPRs for the ROC curve, importing the metrics module if needed.

from sklearn import metrics

fpr, tpr, thresholds = metrics.roc\_curve(y\_test, pos\_proba)

Now we need to produce a plot. We'll use plt.plot, which will make a line plot using the first argument as the x values (FPRs), the second argument as the y values (TPRs), and the shorthand '\*-' to indicate a line plot with star symbols where the data points are located. We add a straight-line plot from the point (0, 0) to (1, 1), which will appear in red ('r') and as a dashed line ('--'). We've also given the plot a legend (which we'll explain shortly), as well as axis labels and a title. This code produces the ROC plot:

plt.plot(fpr, tpr, '\*-')

plt.plot([0, 1], [0, 1], 'r--')

plt.legend(['Logistic regression', 'Random chance'])

plt.xlabel('FPR')

plt.ylabel('TPR')

plt.title('ROC curve')

And the plot should look like this:

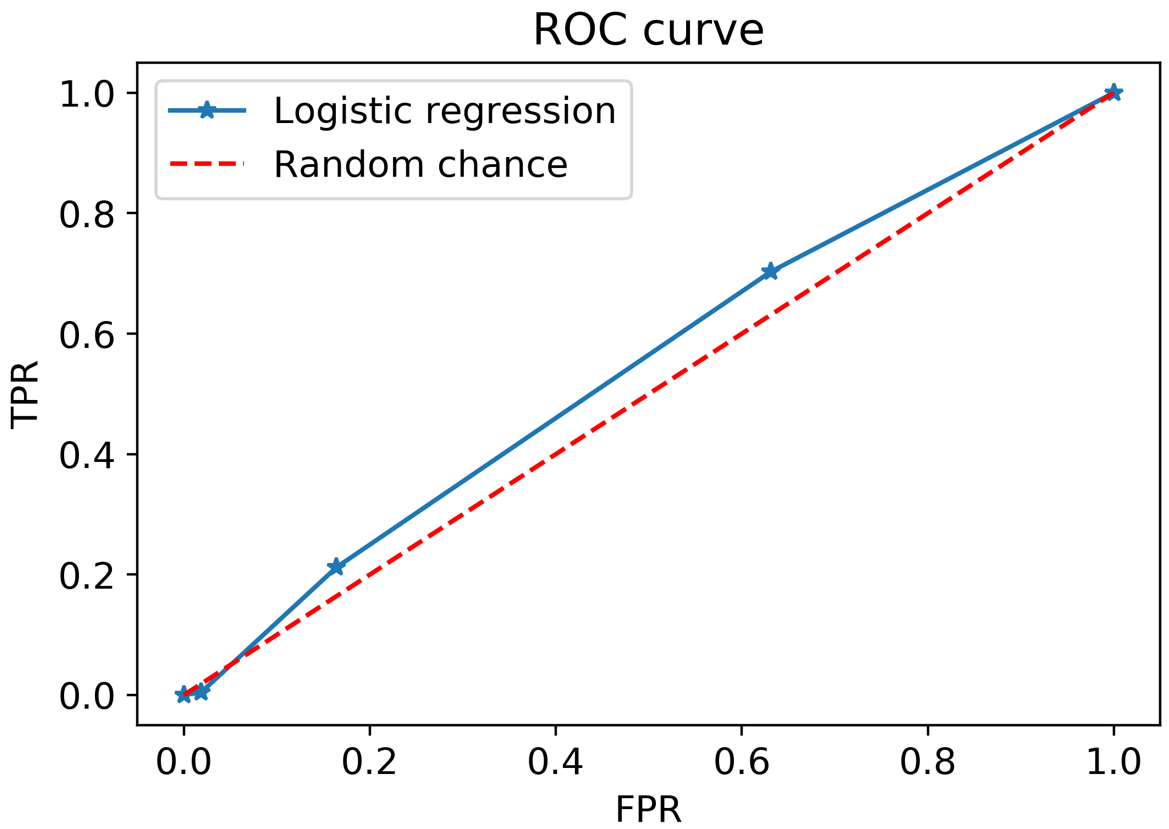


Figure 2.29: ROC curve for our logistic regression, with a line of random chance   
shown for comparison

What have we learned from our ROC curve? We can see that it starts at (0,0) with a threshold high enough so that there are no positive classifications. Then the first thing that happens, as we imagined previously when lowering the threshold to about 0.25, is that we get an increase in the false positive rate, but very little increase in the true positive rate. The effects of continuing to lower the threshold so that the other bars from our stacked histogram plot in Figure 2.28would be included as positive classifications are shown by the subsequent points on the line. We can see the thresholds that lead to these rates by examining the threshold array, which is not part of the plot. View the thresholds used to calculate the ROC curve using this code:

thresholds

The output should be as follows:

array([1.2549944 , 0.2549944 , 0.24007604, 0.22576598, 0.21207085])

Notice that the first threshold is actually above 1; practically speaking, it just needs to be a threshold that's high enough that there are no positive classifications.

Now consider what a "good" ROC curve would look like. As we lower the threshold, we want to see the TPR increase, which means our classifier is doing a good job of correctly identifying positive samples. At the same time, ideally the FPR should not increase that much. The ROC curve of an effective classifier would "hug" the upper left corner of the plot: high true positive rate, low false positive rate. You can imagine that a perfect classifier would get a TPR of 1 (recovers all the positive samples) with an FPR of 0 and appear as a sort of square starting at (0,0), going up to (0,1), and finishing at (1,1). While in practice this kind of performance is highly unlikely, it gives us a limiting case.

Further consider what the area under the curve (AUC) of such a classifier would be, remembering integrals from calculus if you have taken it. The AUC of a perfect classifier would be 1, because the shape of the curve would be a square on the unit interval [0, 1].

On the other hand, the line labeled as "Random chance" in our plot is the ROC curve that theoretically results from flipping an unbiased coin as a classifier: it's just as likely to get a true positive as a false positive, so lowering the threshold introduces more of each in equal proportion and the TPR and FPR increase at the same rate. The AUC under this ROC would be half of the perfect classifier's, as you can see graphically, and would be 0.5.

So, in general, the ROC AUC is going to be between 0.5 and 1 (although values below 0.5 are technically possible). Values close to 0.5 indicate the model can do little better than random chance (coin flip) as a classifier, while values closer to 1 indicate better performance. The ROC AUC is a key metric for the quality of a classifier and is widely used in machine learning. The ROC AUC may also be referred to as the C-statistic.

Being such an important metric, naturally scikit-learn has a convenient way to calculate it. Let's see what the ROC AUC of the logistic regression classifier is, where we can pass the same information that we did to the roc\_curve function. Calculate the area under the ROC curve with this code:

metrics.roc\_auc\_score(y\_test, pos\_proba)

And observe the output:

0.5434650477972642

The ROC AUC for the logistic regression is pretty close to 0.5, meaning it's not a very effective classifier. This may not be surprising, considering we have expended no effort to determine which features out of the candidate pool are actually useful, at this point. We're just getting used to model fitting syntax and learning the way to calculate model quality metrics using a simple model containing only the EDUCATION feature. Later on, by considering other features, hopefully we'll get a higher ROC AUC.

Note

ROC curve: How did it get that name?

During World War II, radar receiver operators were evaluated on their ability to judge whether something that appeared on their radar screen was in fact an enemy aircraft or not. These decisions involved the same concepts of true and false positives and negatives that we are interested in for binary classification. The ROC curve was devised as a way to measure the effectiveness of operators of radar receiver equipment.

Precision

Before embarking on the activity, we consider the classification metric briefly introduced previously: precision. Like the ROC curve, this diagnostic is useful over a range of thresholds. Precision is defined as follows:

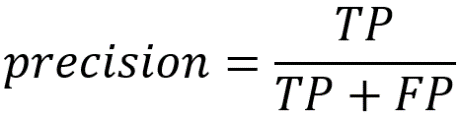


Figure 2.30: Precision equation

Consider the interpretation of this, in the sense of varying the threshold across the range of predicted probabilities, as we did for the ROC curve. At a high threshold, there will be relatively few samples predicted as positive. As we lower the threshold, more and more will be predicted as positive. Our hope is that as we do this, the number of true positives increases more quickly than the number of false positives, as we saw on the ROC curve. Precision looks at the ratio of the number of true positives to the sum of true and false positives. Think about the denominator here: what is the sum of true and false positives?

This sum is in fact the total number of positive predictions, since all positive predictions will be either correct or incorrect. So, precision measures the ratio of positive predictions that are correct, to all positive predictions. For this reason, it is also called positive predictive value. Precision is often used when the classes are very imbalanced and is more focused on getting good model performance in the sense of the correctness of positive predictions. It's easy to get a high true positive rate by predicting many samples as positive, but this also increases the number of false positives relative to the number of positive predictions. If there are very few positive samples, precision gives a more critical assessment of the quality of a classifier than the ROC AUC. As with the ROC curve, there is a convenient function in scikit-learn to calculate precision, together with recall (also known as the true positive rate), over a range of thresholds: metrics.precision\_recall\_curve.

Why might precision be a useful measure of classifier performance? Imagine that for every positive model prediction, you are going to take some expensive course of action, such as a time-consuming review of content that was flagged as inappropriate by an automated analysis. False positives would waste the valuable time of human reviewers. You would want to be sure that you were making the right decisions on what content received detailed review. Precision could be a good metric to use in this situation.

Activity 2: Performing Logistic Regression with a New Feature and Creating a Precision-Recall Curve

In this activity, you'll train a logistic regression using a feature besides EDUCATION. Then you will graphically assess the tradeoff between precision and recall, as well as calculate the area underneath a precision-recall curve. You will also calculate the ROC AUC on both the training and testing sets and compare them.

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 found here: [~~http://bit.ly/2UWPgYo~~](http://bit.ly/2UWPgYo).

1. Use scikit-learn's train\_test\_split to make a new set of training and testing data. This time, instead of EDUCATION, use LIMIT\_BAL: the account's credit limit, as the feature.
2. Train a logistic regression model using the training data from your split.
3. Create the array of predicted probabilities for the testing data.
4. Calculate the ROC AUC using the predicted probabilities and the true labels of the testing data. Compare this to the ROC AUC from using the EDUCATION feature.
5. Plot the ROC curve.
6. Calculate the data for the precision-recall curve on the testing data using scikit-learn functionality.
7. Plot the precision-recall curve using matplotlib.
8. Use scikit-learn to calculate the area under the precision-recall curve
9. Now recalculate the ROC AUC, except this time do it for the training data. How is this different, conceptually and quantitatively, from your earlier calculation?

The solution to this activity can be found on page ~~331~~.

Summary

In this chapter, we finished the initial exploration of the case study data by examining the response variable. Once we became confident in the completeness and correctness of the data set, we were prepared to explore the relation between features and response and build models.

We spent much of this chapter getting used to model fitting in scikit-learn at a technical, coding level, and learning about metrics we could use with the binary classification problem of the case study. When trying different feature sets and different kinds of models, you will need some way to tell if one approach is working better than another. Consequently, you'll need to use model performance metrics like those we learned in this chapter.

While accuracy is a familiar and intuitive metric as the percentage of correct classifications, we learned why it may not give a useful assessment of the performance of a classifier. We learned how to use a majority class null model to tell whether an accuracy rate is truly good, or no better than what would result from simply predicting the most common class for all samples. When the data are imbalanced, accuracy is usually not the best way to judge a classifier.

In order to have a more nuanced view of how a model is performing, it's necessary to separate out the positive and negative classes and assess the accuracy on these independently. From the resulting counts of true and false positive and negative classifications, which can be summarized in a confusion matrix, we can derive several other metrics: true and false positive and negative rates. Combining true and false positives and negatives with the concept of predicted probabilities and a variable threshold of prediction, we can further characterize the usefulness of a classifier using the ROC curve, the precision-recall curve, and the areas under these curves.

With these tools, you are well equipped to answer general questions about the performance of a binary classifier, in any domain you may be working in. Later in the book, we will learn about application-specific ways to assess model performance, by attaching costs and benefits to true and false positives and negatives. Before that, starting in the next chapter, we will begin learning the details behind what is possibly the most popular and simplest classification model: logistic regression.

Chapter 2: Introduction to Scikit-Learn and Model Evaluation

Activity 2: Performing Logistic Regression with a New Feature and Creating a Precision-Recall Curve

1. Use scikit-learn's train\_test\_split to make a new set of training and testing data. This time, instead of EDUCATION, use LIMIT\_BAL: the account's credit limit, as the feature.

Execute the following code to do this:

X\_train\_2, X\_test\_2, y\_train\_2, y\_test\_2 = train\_test\_split(

df['LIMIT\_BAL'].values.reshape(-1,1),

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

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

Notice here we create new training and testing splits, with new variable names.

1. Train a logistic regression model using the training data from your split.

The following code does this:

example\_lr.fit(X\_train\_2, y\_train\_2)

You can reuse the same model object you used earlier, example\_lr, if you’re running the whole chapter in a single notebook. You can re-train this object to learn the relationship between this new feature and the response. You could even try a different train/test split, if you wanted to, without creating a new model object. The existing model object has been updated in-place in these scenarios.

1. Create the array of predicted probabilities for the testing data:

Here is the code for this step:

y\_test\_2\_pred\_proba = example\_lr.predict\_proba(X\_test\_2)

1. Calculate the ROC AUC using the predicted probabilities and the true labels of the testing data. Compare this to the ROC AUC from using the EDUCATION feature:

Run this code for this step:

metrics.roc\_auc\_score(y\_test\_2, y\_test\_2\_pred\_proba[:,1])

The output is as follows:

0.6201990844642832

Notice we index the predicted probabilities array in order to get the predicted probability of the positive class from the second column. How does this compare to the ROC AUC from the EDUCATION logistic regression? The AUC is higher. This may be because now we are using a feature that has something to do with an account's financial status (credit limit), to predict something else related to the account's financial status (whether or not it will default), instead of using something less directly related to finances.

1. Plot the ROC curve.

Here is the code to do this; it's similar to the code we used in the previous exercise:

fpr\_2, tpr\_2, thresholds\_2 = metrics.roc\_curve(

y\_test\_2, y\_test\_2\_pred\_proba[:,1])

plt.plot(fpr\_2, tpr\_2, '\*-')

plt.plot([0, 1], [0, 1], 'r--')

plt.legend(['Logistic regression', 'Random chance'])

plt.xlabel('FPR')

plt.ylabel('TPR')

plt.title('ROC curve for logistic regression with LIMIT\_BAL feature')

The plot should appear as follows:

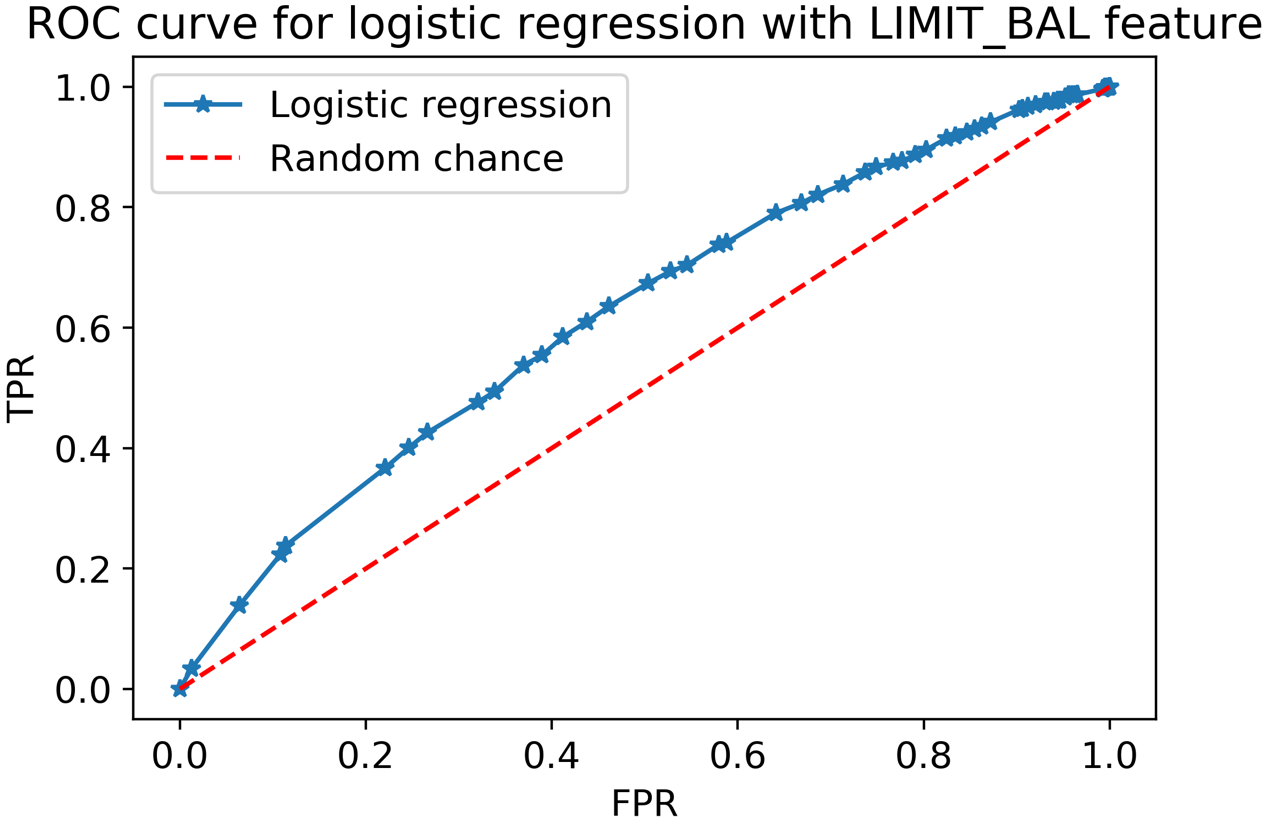


Figure 6.48: ROC curve for the LIMIT\_BAL logistic regression

This looks a little closer to an ROC curve that we'd like to see: it's a bit further from the random chance line than the model using only EDUCATION. Also notice that the variation in pairs of true and false positive rates is a little smoother over the range of thresholds, reflective of the larger number of distinct values of the LIMIT\_BAL feature.

1. Calculate the data for the precision-recall curve on the testing data using scikit-learn functionality.

Precision is often considered in tandem with recall. You are already familiar with recall. This is just another word for the true positive rate. We can use precision\_recall\_curve in sklearn.metrics to automatically vary the threshold and calculate pairs of precision and recall values at each one. Here is the code to retrieve these values, which is similar to roc\_curve:

precision, recall, thresh\_3 = \

metrics.precision\_recall\_curve(y\_test\_2, y\_test\_2\_pred\_proba[:,1])

1. Plot the precision-recall curve using matplotlib: we can do this with the following code.

Note that we put recall on the x-axis, precision on the y-axis, and set the axes limits to the range [0, 1]:

plt.plot(recall, precision, '-x')

plt.xlabel('Recall')

plt.ylabel('Precision')

plt.title(Precision and recall for the logistic regression with LIMIT\_BAL')

plt.xlim([0, 1])

plt.ylim([0, 1])

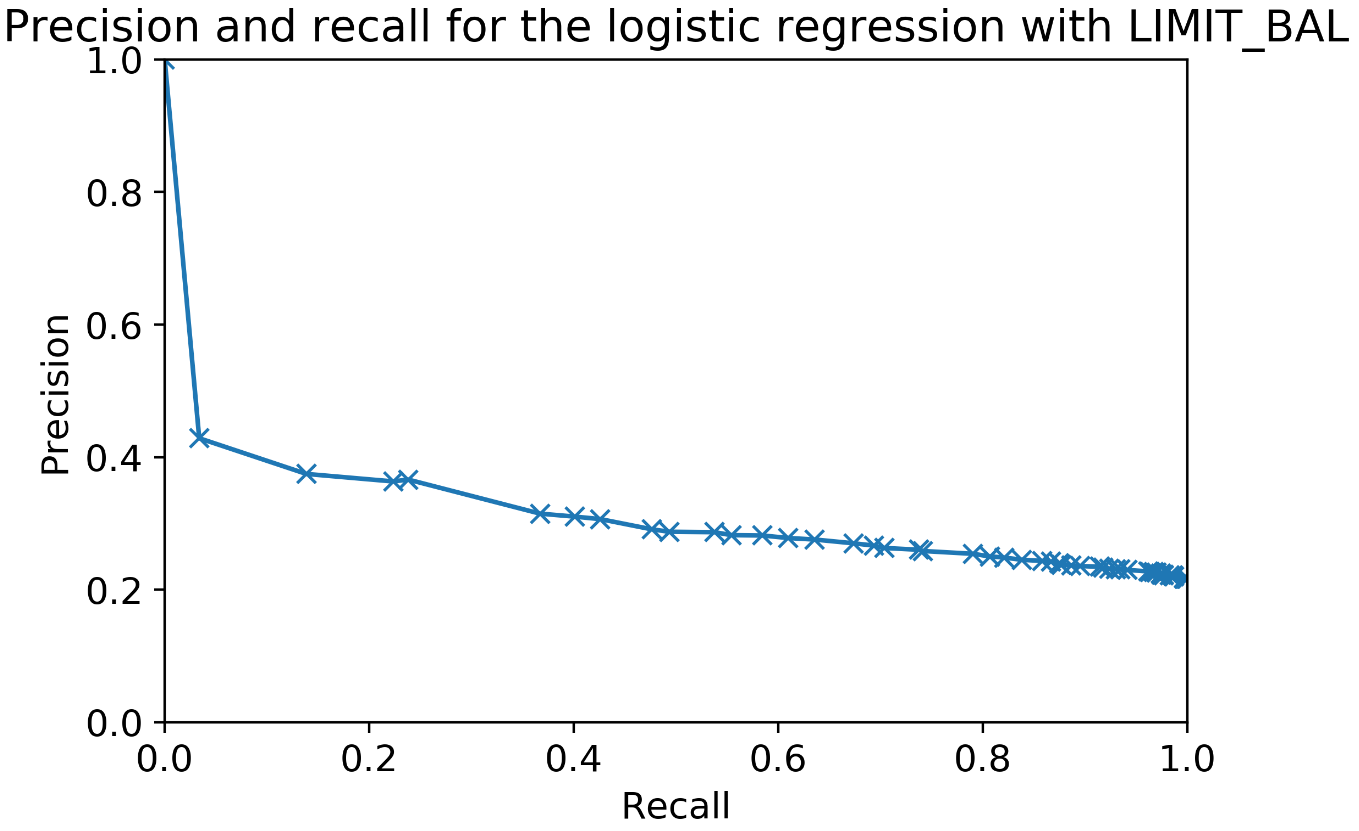


Figure 6.49: Plot of the precision-recall curve

1. Use scikit-learn to calculate the area under the precision-recall curve.

Here is the code for this:

metrics.auc(recall, precision)

You will obtain the following output:

0.31566964427378624

We saw that the precision-recall curve shows that precision is generally fairly low for this model; for nearly all of the range of thresholds, the precision, or portion of positive classifications that are correct, is less than half. We can calculate the area under the precision-recall curve as a way to compare this classifier with other models or feature sets we may consider.

Scikit-learn offers functionality for calculating an area under the curve for any set of x-y data, using the trapezoid rule, which you may recall from calculus: metrics.auc. We used this functionality to get the area under the precision-recall curve.

1. Now recalculate the ROC AUC, except this time do it for the training data. How is this different, conceptually and quantitatively, from your earlier calculation?

First, we need to calculate predicted probabilities using the training data, as opposed to the testing data. Then we can calculate the ROC AUC using the training data labels. Here is the code:

y\_train\_2\_pred\_proba = example\_lr.predict\_proba(X\_train\_2)

metrics.roc\_auc\_score(y\_train\_2, y\_train\_2\_pred\_proba[:,1])

You should obtain the following output:

0.6182918113358344

Quantitatively, we can see that this AUC is not all that different from the testing data ROC AUC we calculated earlier. Both are about 0.62. Conceptually, what is the difference? When we calculate this metric on the training data, we are measuring the model's skill in predicting the same data that "taught" the model how to make predictions. We are seeing how well the model fits the data. On the other hand, testing data metrics indicate performance on out-of-sample data the model hasn’t “seen” before. If there was much of a difference in these scores, which usually would come in the form of a higher training score than testing score, it would indicate that while the model fits the data well, the trained model does not generalize well to new, unseen data.

In this case, the training and testing scores are similar, meaning the model does about as well on out-of-sample data as it does on the same data used in model training. We will learn more about the insights we can gain by comparing training and testing scores in Chapter 4, The Bias-Variance Trade-off.