5 Fitting a logistic regression

This chapter covers

- Model fitting
- Model interpretation and evaluation
- Classification metrics
- Data exploration through histograms and correlation heat maps

Logistic regression is a supervised learning method for predicting a binary response from one or more independent variables. It's commonly used for classification tasks where the dependent variable represents two possible categories or classes (e.g., pass or fail, presence or absence). It estimates the probability that a given instance belongs to a particular category based on the values of the independent variables, or predictors, using the logistic function (also known as the sigmoid function) that maps the output to a range between 0 and 1. We'll see how the value between 0 and 1 translates to binary outcomes in section 5.1.

The use cases are infinite. Here is just a small sample of problems that can be solved with logistic regression:

- Banks predicting whether a loan applicant will default, using factors such as credit score, credit history, income, and (if allowed) demographic data
- Wireless carriers predicting the likelihood of customers canceling their service, based on usage patterns and satisfaction scores
- Political scientists predicting election outcomes by analyzing survey data
- Meteorologists predicting the probability of rain from a mix of satellite and radar data, atmospheric humidity, cloud cover, and other factors

The results are immediately actionable. Banks, for instance, will reject a loan application if the probability of defaulting is greater than 50% or, alternatively, extend an amount of credit if that same probability is less than 50%.

When fitting a logistic regression, we are attempting to answer a series of questions previously applied to linear regression in chapter 4, but specifically focused on predicting binary outcomes:

- *Can it predict the probability of a dependent variable equaling yes (e.g., a customer will churn, a patient has heart disease)?* Logistic regression returns coefficients that can be plugged into an equation to predict binary outcomes.
- *Is there a meaningful relationship between the dependent variable and the one or more independent variables?* Logistic regression identifies which predictors have, and don't have, a statistically significant effect on the probability of a binary outcome.
- *How can it measure the relationship between predictors and outcomes?* Logistic regression provides quantitative insights into how changes in the predictors affect the probability of the binary outcome.
- *How does it assess the strength and direction of associations between variables?* The returned coefficient estimates measure the relationships between predictors and the log odds of the binary outcome.

Our journey will be very similar to the path blazed in the previous chapter. We'll get into the nuts and bolts of logistic regression, sometimes by comparing and contrasting it to linear modeling; we'll import a real data set and immediately go about analyzing and visualizing it, variable by variable; and then we'll demonstrate how to fit a multiple logistic regression, interpret and evaluate the results, apply the coefficient estimates to make binary predictions, and compute a set of classification metrics to further evaluate the predictive power of the model.

We'll be attempting to solve a rather innocuous classification problem: correctly predicting the variety of raisins from a set of physical properties. However, the methods demonstrated throughout this chapter can be applied to *any* problem, on *any* data set, across *any* domain—as long as the dependent variable is binary. Let's begin our journey with a deep dive into all things logistic regression.

5.1 Logistic regression vs. linear regression

Logistic regression is a supervised learning method used to model the relationship between a binary dependent variable and one or more independent variables, or predictors. A simple logistic regression is when the binary dependent variable is regressed against just one predictor, which can be numeric or categorical; a multiple logistic regression, on the other hand, is when the dependent variable is regressed against two or more predictors. In the previous chapter, we fit a simple regression; this time, we'll demonstrate how to fit a multiple regression.

In logistic regression, the dependent variable, which is typically coded as 0 or 1, is modeled as a function of the independent variables using the *logistic function*, also known as the *sigmoid function*. This function transforms the linear combination of the predictors into a probability for each record in the data that equals some value between 0 and 1. When the probability equals 50% or greater, the model is predicting the binary outcome to equal 1; conversely, when the probability is less than 50%, the model is instead predicting the binary outcome to equal 0.

The sigmoid function is typically written in its standard form as

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This formulation is useful for visualizing the S-shaped curve, as shown in figure 5.1, where x is treated as a single continuous variable.

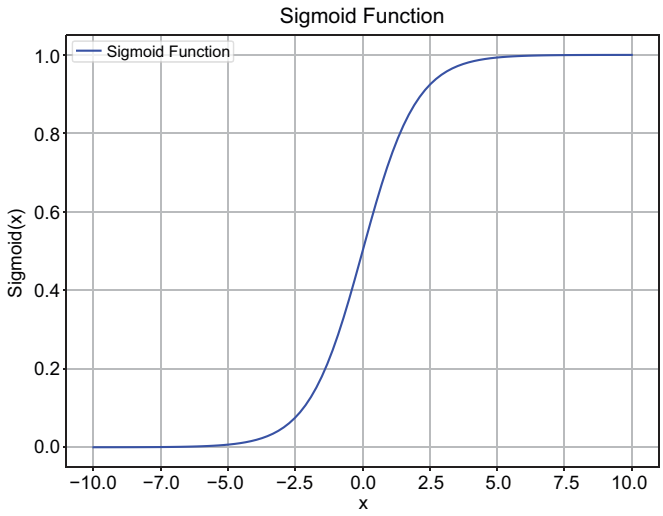

Figure 5.1 A graphical illustration of the sigmoid function. The sigmoid function is characterized by its S-shaped curve, compared to a linear function, which is characterized by a straight line.

We get probabilities by plugging the predictors and their coefficient estimates into this function, where the input is replaced by a linear combination of predictors:

$$P(Y = 1|X) = \frac{1}{1 + e^{-(\beta_0+\beta_1 X_1+\beta_2 X_2+ \beta_3 X_3)}}$$

where

- $P(Y = 1|X)$ is the probability of the outcome variable, designated as Y, being 1 given the predictor variables X.
- e is the base of the natural logarithm, equal to 2.72.
- β_0 is the intercept term.
- β_1, β_2, and β_3 are the coefficients of the predictor variables.
- $X1$, $X2$, and $X3$ are the predictor variables.

In linear regression we are trying to predict changes in a numeric dependent variable given the predictors; but in logistic regression, we are instead applying the sigmoid function to predict the probability of a binary dependent variable coded as `0` or `1` to be 1 from a single predictor in a simple regression or from two or more predictors in a multiple regression. Whereas the coefficients in linear modeling represent the change in the dependent variable for a one-unit change in the independent variable, the coefficient estimates in logistic regression represent the change in the log odds of the binary outcome for a one-unit change in the independent variable.

The sigmoid function, usually denoted as $\sigma(x)$, is a mathematical function that maps any real number to a value between 0 and 1. When the input x is large, either positive or negative, the value of e^{-x} approaches 0, causing the denominator of the fraction to become large, thereby resulting in $\sigma(x)$ being close to 1. Conversely, when x is large in the negative direction, e^{-x} subsequently becomes large, causing $\sigma(x)$ to approach 0.

Now that we have a good theoretical understanding of logistic regression and are aware of the differences between it and linear regression, we are prepared to take the next steps. We'll import a real data set, explore it, demonstrate how to fit a multiple logistic regression, explain how to interpret and evaluate and apply the model output, and show how to compute classification metrics to assess the model's accuracy and predictive power.

5.2 Multiple logistic regression

Our plan is to import a real data set that is both longer and wider than the data we worked with in the previous chapter. It contains a single binary variable (which will be our dependent variable) and several numeric variables (which will be our predictors). As mentioned earlier, our data set deals with raisins. Two varieties of raisins are grown in Turkey: Kecimen and Besni. Our data contains the morphological features for 450 Kecimen raisins and 450 Besni raisins. Morphological features typically refer to the size, shape, and other physical qualities of objects or other entities, usually in the

context of image processing. In fact, the numeric variables are measured in pixels or derived from the pixel count (pixels represent the smallest unit of measurement in digital imaging). We'll discover that Kecimen and Besni raisins have several distinguishing qualities, which provides some hope that a logistic regression can accurately predict which raisin is of what variety given their morphological features.

Once we have imported the data, we'll analyze it in ways that go above and beyond the exploratory data analysis performed in chapter 4. Then we'll demonstrate how to fit a logistic regression model, how to digest the output, and how to calculate other measures that go a long way toward quantifying model accuracy and predictive power.

Data exploration is an exercise that typically involves a mix of basic statistical and visualization techniques to reveal patterns, trends, and relationships that might influence the scope and direction of more advanced methods. When preparing to fit a regression model, linear or logistic, exploratory data analysis helps us understand basic properties of each variable, such as counts for categorical variables or distributions for numeric variables; identify missing values and detect outliers; reveal relationships between variables; and even drive variable selection. Because we intend to fit a logistic regression with a binary dependent variable and numeric predictors, our preliminary analysis of the data will mostly focus on understanding the similarities and differences in the characteristics of every predictor segmented by category.

5.2.1 Importing and exploring the data

Our data set is a .csv file stored in our working directory. We read it into Python by importing the `pandas` library and then making a call to the `pd.read_csv()` method. The following line of code imports raisin_dataset.csv into a data frame called `raisins`:

```
>>> import pandas as pd
>>> raisins = pd.read_csv('raisin_dataset.csv')
```

Now that we've imported our data, we can call other methods to analyze and visualize it.

Understanding the data

The `info()` method in `pandas` returns a concise summary of the `raisins` data frame. A call to `info()` is a quick, easy, and logical starting point for any data exploration exercise:

```
>>> print(raisins.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 900 entries, 0 to 899
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Area             900 non-null    int64
 1   MajorAxisLength  900 non-null    float64
 2   MinorAxisLength  900 non-null    float64
 3   Eccentricity     900 non-null    float64
 4   ConvexArea       900 non-null    float64
```

```
 5   Extent           900 non-null    float64
 6   Perimeter        900 non-null    float64
 7   Class            900 non-null    object
dtypes: float64(6), int64(1), object(1)
memory usage: 56.4+ KB
None
```

This tells us the following about the data:

- *The dimensions*—It's 900 × 8, or 900 rows by 8 columns.
- *The column names and types*—The morphological features are represented by seven integers and floats, and the one binary variable is `Class`.
- *The number of non-null values by variable*—This equals 0 across the width of the data frame.

The morphological features are defined as follows:

- *Area* represents the number of pixels within the boundary of each raisin.
- *Major axis length* is the length of the main axis, which is the longest line that can be drawn on the raisin grain.
- *Minor axis length* is the length of the small axis, which is the shortest line that can be drawn on the raisin grain.
- *Eccentricity* is a measure of how elongated or stretched a raisin grain is compared to an ellipse.
- *Convex area* represents the number of pixels within the total area enclosed by the outer perimeter of the raisin grain.
- *Extent* gives the ratio of the region formed by the raisin grain to the total pixels in the bounding box.
- *Perimeter* is a measure of the distance between the boundaries of the raisin grain and the pixels around it.

Of course, it's always helpful to view at least some of the content instead of just being aware of its dimensions and other properties; we'll next show how to best go about doing that, considering the length and width of the data.

VIEWING THE DATA

In the prior chapter, because we were working with a data frame that was just 20 rows long and 2 columns wide, it was practical to print the whole object. That's no longer an option. We first make a call to the `set_option()` method from the `pandas` library to instruct Python to print every column in the console, rather than just a few, for easy fit:

```
>>> pd.set_option('display.max_columns', None)
```

Then we pass our data frame to the `head()` and `tail()` methods, which return the first three and last three records, respectively. The `head()` and `tail()` methods will print whatever number of records are passed (although 10 or fewer should be sufficient for most use cases):

```
>>> print(raisins.head(n = 3))
    Area MajorAxisLength MinorAxisLength Eccentricity ConvexArea Extent
0  87524          442.25          253.29         0.82    90546.0   0.76
1  75166          406.69          243.03         0.80    78789.0   0.68
2  90856          442.27          266.33         0.80    93717.0   0.64
   Perimeter    Class
0    1184.04  Kecimen
1    1121.79  Kecimen
2    1208.58  Kecimen

>>> print(raisins.tail(n = 3))
     Area MajorAxisLength MinorAxisLength Eccentricity ConvexArea
897 99657          431.71          298.84         0.72   106264.0
898 93523          476.34          254.18         0.85    97653.0
899 85609          512.08          215.27         0.91    89197.0
    Extent Perimeter Class
897   0.74   1292.83 Besni
898   0.66   1258.55 Besni
```

As previously mentioned, the binary dependent variable is typically coded as `0` or `1` before regressing it against one or more predictors. However, it's unlikely the raw data will ever be coded that way. So, you absolutely need to know how to transform the values, in this case from Kecimen and Besni, to `0` and `1`. It doesn't matter which is which—as long as you keep in mind that when we fit our logistic regression, we're predicting the probability of the binary outcome equaling `1`.

Transforming the original values requires just two lines of Python code. In the first, we map the categorical labels `Kecimen` and `Besni` to numerical form to subsequently convert them to equal `0` and `1`, respectively:

```
>>> class_mapping = {'Kecimen': 0, 'Besni': 1}
```

In the second line of code, we call the `map()` method to map the values in the variable `Class` using the previously defined mapping dictionary. This line then replaces the original class labels, `Kecimen` and `Besni`, with their corresponding numerical values, `0` and `1`, effectively converting the variable `Class` from categorical to numerical (but still binary):

```
>>> raisins['Class'] = raisins['Class'].map(class_mapping)
```

Let's confirm the success of this operation by rerunning the `head()` and `tail()` methods. This time, however, we add instructions to both methods so that `head()` and `tail()` return just a subset of the `raisins` variables. It's not necessary, after all, to again view the entire width of the `raisins` data frame; we merely want to confirm that the variable `Class` is now in numeric form:

```
>>> print(raisins[['Area', 'MajorAxisLength', 'Class']].head(n = 3))
    Area  MajorAxisLength  Class
0  87524          442.25      0
1  75166          406.69      0
2  90856          442.27      0
```

```
>>> print(raisins[['Area', 'MajorAxisLength', 'Class']].tail(n = 3))
      Area  MajorAxisLength  Class
897  99657           431.71      1
898  93523           476.34      1
899  85609           512.08      1
```

And it is—where `Class` equaled `Kecimen`, it now equals `0`, and where `Class` equaled `Besni`, it now equals `1`.

COMPUTING BASIC STATISTICS

In the previous chapter, we called the `describe()` method to return a series of descriptive statistics on two numeric variables. But now it makes much more sense to pull some of those same statistics divided by `Kecimen` versus `Besni`, or `0` versus `1`. So, rather than make another call to `describe()`, we instead call the `agg()` and `groupby()` methods to return the minimum (`min`), maximum (`max`), `mean`, `median`, standard deviation (`std`), and variance (`var`) from the numeric variables, grouped by the variable `Class`. Instructions are then provided to limit the results to two decimal places. Although `round(2)` is applied to limit results to two decimal places, large values may nevertheless appear in scientific notation due to `pandas` display defaults:

```
>>> basic_statistics = (raisins.groupby('Class') \
>>>                     .agg(['min', 'max', 'mean', 'median',
>>>                           'std', 'var']))
>>> print(basic_statistics.round(2))
      Area
         min     max       mean    median       std           var
Class
0      25387  180898   63413.47   61420.0  17727.77  3.142738e+08
1      40702  235047  112194.79  104426.5  39229.90  1.538985e+09
      MajorAxisLength
                  min     max    mean  median     std       var
Class
0              225.63  843.96  352.86  350.24   59.61   3553.54
1              274.17  997.29  509.00  493.19  105.77  11187.53
      MinorAxisLength
                  min     max    mean  median    std      var
Class
0              143.71  326.90  229.35  228.62  34.06  1159.96
1              172.51  492.28  279.62  273.36  50.76  2576.98
      Eccentricity
                  min   max   mean  median    std    var
Class
0                0.35  0.92   0.74    0.77   0.09   0.01
1                0.50  0.96   0.82    0.83   0.07   0.00
      ConvexArea
           min       max   mean      median       std            var
Class
0      26139.0  221396.0   65696.36 63826.5   19005.89  3.612239e+08
1      41924.0  278217.0  116675.82 108062.5  40797.07  1.664401e+09
```

```
      Extent
             min   max   mean     median       std    var
Class
0           0.45  0.84   0.71       0.71      0.04    0.0
1           0.38  0.83   0.69       0.70      0.06    0.0

      Perimeter
            min      max     mean   median     std       var
Class
0        619.07  2253.56   983.69   977.93  150.31  22591.74
1        771.80  2697.75  1348.13  1305.80  246.80  60912.53
```

Let's briefly review these measures:

- The *minimum* is the smallest observed value.
- The *maximum* is the largest observed value.
- The *mean* represents the sum of all values divided by the number of records.
- The *median* represents the middle value when the data is sorted in ascending order. When the number of observations equals an even number, the median is derived by taking the average, or mean, of the two middle values. It is equivalent to the second quartile.
- The *standard deviation* quantifies the amount of variation, or dispersion, from the mean.
- So does the *variance*; it can be derived by squaring the standard deviation.

From these descriptive statistics, we can gather that Kecimen and Besni raisins generally have very different physical qualities: Kecimen raisins tend to be smaller and rounder compared to the Besni variety, whereas Besni raisins are larger and more elongated. These distinguishable qualities suggest that at least some of these variables might be good predictors of class labels. However, if the variances in these same measures are significant enough to negate some of the distinguishing characteristics that separate the two varieties of raisins, it could be an indication that the segmentation is far from perfect, which might result in misclassifications. We'll make a best attempt to flush out these similarities and dissimilarities by drawing a series of histograms.

Histograms

A *histogram* is a graphical representation of the shape, central tendency, and dispersion of numeric data. It contains a series of adjacent bars, where the width of each bar corresponds to a specific range of values, represented on the *x* axis, and the height of each bar corresponds to the number of occurrences, represented on the *y* axis. A paired histogram displays two distributions—the same numeric data series, but split by class—so that their respective shapes can be readily compared and contrasted.

If Kecimen and Besni raisins have variances that indicate partial similarities in their respective distributions across the morphological features in our data, those will be revealed by drawing a series of paired histograms (see figure 5.2). Rather than create seven paired histograms from seven similar snippets of code, we'll create a

`matplotlib` grid of paired histograms by iterating through each numeric variable in the `raisins` data frame:

```
>>> import matplotlib.pyplot as plt
>>> fig, ((ax1, ax2, ax3), (ax4, ax5, ax6),
>>>       (ax7, ax8, ax9)) = plt.subplots(nrows = 3,
>>>                                       ncols = 3,
>>>                                       figsize = (8, 10))

>>> axes = [ax1, ax2, ax3, ax4, ax5,
>>>         ax6, ax7, ax8, ax9]
>>> columns = ['Area', 'MajorAxisLength', 'MinorAxisLength',
>>>            'Eccentricity', 'ConvexArea',
>>>            'Extent', 'Perimeter']

>>> for i, column in enumerate(columns):
>>>     for class_label,
>>>     data in raisins.groupby('Class'): \
>>>         data[column] \
>>>         .hist(alpha = 0.5,
>>>      label = class_label,
>>>              ax = axes[i])
>>>     axes[i].set_title(f'{column}')
>>>     axes[i].set_xlabel(column)
>>>     axes[i].set_ylabel('Frequency')
>>>     axes[i].legend(labels = ['Kecimen',
>>>                              'Besni'])

>>> for ax in [ax8, ax9]:
>>>     ax.axis('off')

>>> plt.tight_layout()
>>> plt.show()
```

Libraries and modules must be imported (once per script) before running subsequent code.

Creates a 3 × 3 grid of subplots ax1 through ax9 that, when displayed, will be 8 × 10 in size

Creates a list of subplots ax1 through ax9

Creates a list of column names to be plotted

Initiates a loop that iterates over each column name

Initiates a loop that iterates over each unique value in Class

Plots the paired histograms, which are configured to be transparent to show overlapping shapes

Sets the title for each histogram to equal the column name

Sets the x-axis label for each histogram to also equal the column name

Sets the y-axis label for each histogram

Establishes a common legend across histograms

Initiates a loop over plots ax8 and ax9 (which actually don't exist)

Hides subplots ax8 and ax9; otherwise, the 3 × 3 grid would contain two empty plots

Displays the grid

Provides vertical separation between adjacent plots to prevent elements from overlapping

Figure 5.2 shows the result.

Drawing paired histograms, especially in preparation for fitting a logistic regression model, provides an opportunity to visually inspect the distributions of the predictors across both classes. By comparing the distributions side by side, we can immediately assess whether there are similarities or differences in the respective distributions between classes. They could reveal which of the variables might be statistically significant predictors for predicting one binary outcome versus the other and therefore guide the variable selection process. Furthermore, paired histograms reveal separation, overlap, or some combination thereof between classes, which is crucial for understanding the potential discriminatory power of the variables and the feasibility of classification through logistic regression.

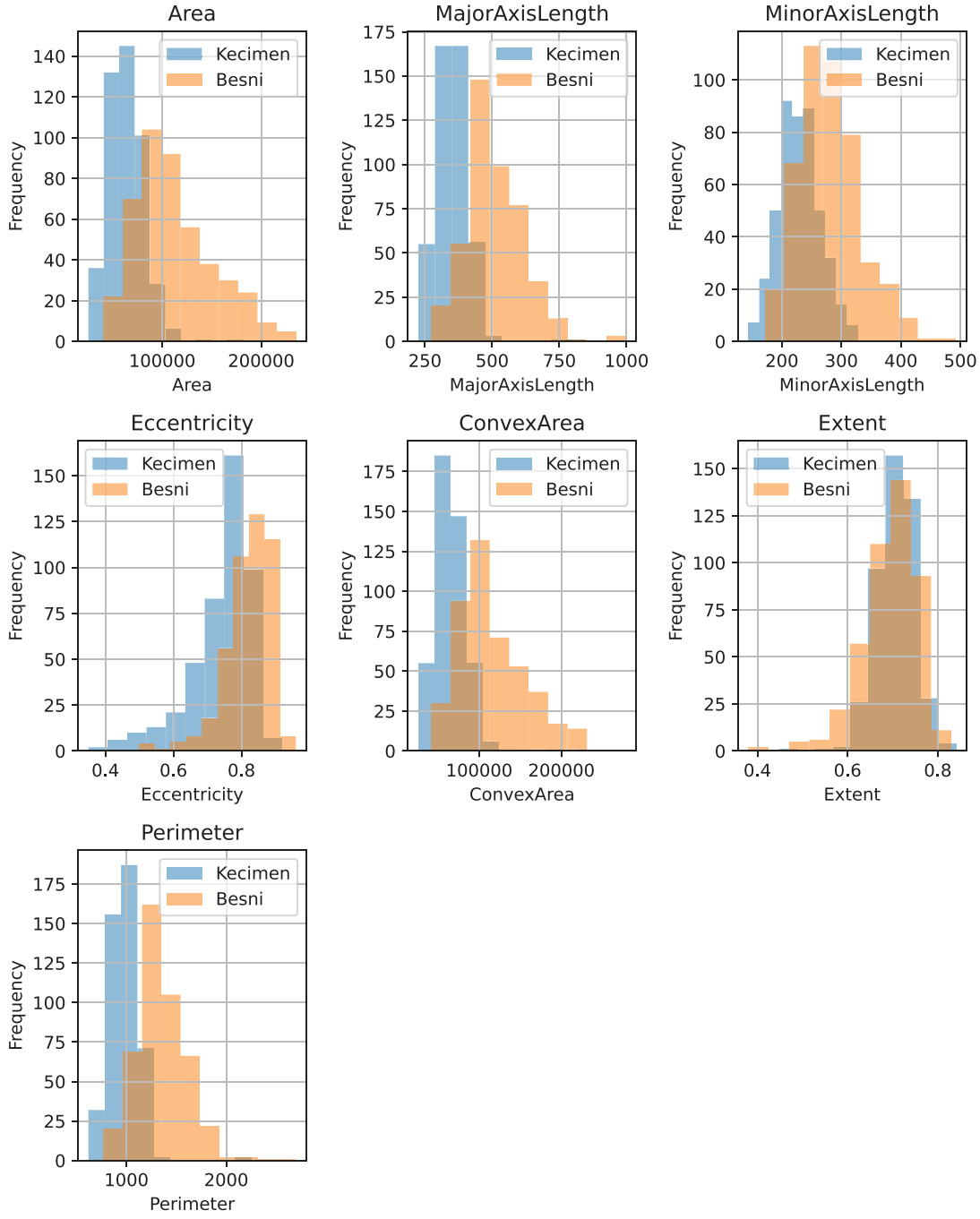

Figure 5.2 A grid of histograms that displays the data distributions of each numeric variable in the `raisins` data frame, further segmented by the binary variable `Class`.

Our plots show a mix of opportunities and challenges. Take `MajorAxisLength` and `Perimeter`, for instance: because their respective distributions between classes are mostly (although not entirely) distinct, their potential value as predictors is substantial. But also consider `Eccentricity` and `Extent`: because their respective distributions between classes are more similar than dissimilar, we may be better off excluding them as predictors.

CORRELATION HEAT MAPS

A *correlation heat map* is a graphical representation of the pairwise correlations between numeric variables. It uses color coding to indicate both the strength and the direction of the correlations. Typically, darker colors, usually deep shades of red and blue, represent strong correlations, whereas lighter shades of the same colors indicate weaker or no correlations. The correlation coefficients are usually added by default. The correlation coefficient between a pair of numeric variables will always equal some number between –1 and +1; the closer the coefficient is to –1 or +1, the stronger the relationship. Two variables are positively correlated when they both move in the same direction, up or down, whereas a pair of variables are negatively correlated when they move in opposite directions.

When fitting a multiple linear regression, understanding the correlations between the predictors and the numeric dependent variable could drive variable selection; that's because independent variables strongly correlated with the response variable, positively or negatively, are the most promising predictors. It's also important to understand the correlation coefficients between the independent variables; strong correlations between potential predictors might be a leading indicator of multicollinearity and therefore a potential loss in predictive power. This is because multicollinearity makes it difficult to isolate the individual effect of each predictor: if two variables are highly correlated, the model can't determine which one is truly influencing the outcome, which inflates standard errors and reduces the statistical significance of meaningful predictors.

In logistic regression, because the dependent variable is binary and not numeric, we can't be concerned about its relationships with the other variables, at least from a correlation perspective. However, we can, and should, be curious about the correlation coefficients between the independent variables. Multicollinearity is a potential issue in logistic regression, just as it is in linear regression. Multicollinearity occurs when independent variables in a regression model are strongly correlated with each other, which can cause difficulties in isolating the effects of individual predictors on the dependent variable and even reduce the predictive power of the model.

That all being said, as a final analysis step before fitting our logistic regression, we'll demonstrate how to create a Matplotlib correlation heat map—actually two maps, one where the data is subset on the variable `Class` equaling `0` and another where the data is subset on `Class` equaling `1`. These will be displayed as one object, printed vertically.

Because we intend to create two correlation heat maps rather than just one, we first split the `raisins` data frame into two equal halves: one where `Class` equals `0` and the other where `Class` equals `1`. `class_0_data` contains those observations from `raisins` where the variety of raisin is Kecimen, and `class_1_data` contains the remaining `raisins` observations where the variety of raisin is Besni. Splitting the data by class makes it possible to perform separate, but equal, analyses on the data:

```
>>> class_0_data = raisins[raisins['Class'] == 0]
>>> class_1_data = raisins[raisins['Class'] == 1]
```

Then we pass our two splits in the data to the `corr()` method, which computes the correlation coefficients between every pair of variables. This method measures only linear relationships, thereby indicating how strongly each pair of variables is linearly related, but it won't capture nonlinear associations:

```
>>> correlation_matrix_class_0 = class_0_data.corr()
>>> correlation_matrix_class_1 = class_1_data.corr()
```

Our pair of correlation heat maps (see figure 5.3) requires `matplotlib` (already imported) and another Python data visualization library called `seaborn`:

```
>>> import seaborn as sns
>>> fig, axs = plt.subplots(2, 1,
>>>                         figsize = (8, 10))
>>> sns.heatmap(correlation_matrix_class_0,
>>>             annot = True,
>>>             cmap = 'coolwarm',
>>>             ax = axs[0])
>>> axs[0].set_title('Correlation Heatmap - \
>>>                   Kecimen Variety')
>>> sns.heatmap(correlation_matrix_class_1,
>>>             annot = True,
>>>             cmap = 'coolwarm',
>>>             ax = axs[1])
>>> axs[1].set_title('Correlation Heatmap - \
>>>                   Besni Variety')
>>> plt.tight_layout()
>>> plt.show()
```

Imports one of the two required libraries

Creates a single figure with two subplots arranged vertically (two rows, one column)

Creates a correlation heat map for the Kecimen variety

Adds the correlation coefficients as annotations to each cell and sets the color map

Sets the title for the first subplot

Creates a correlation heat map for the Besni variety

Adds the correlation coefficients as annotations to each cell and sets the color map

Sets the title for the second subplot

Adjusts the layout to prevent any overlap between subplot

Displays one figure with two heat maps

Figure 5.3 shows the result.

To get the correlation coefficient between any two variables, identify the row and column corresponding to each and then locate where they intersect. So, for instance, where the raisin variety equals Kecimen, the correlation coefficient between `MajorAxisLength` and `MinorAxisLength` equals `0.58`; where the raisin variety is Besni, the correlation coefficient between these same two variables equals `0.62`.

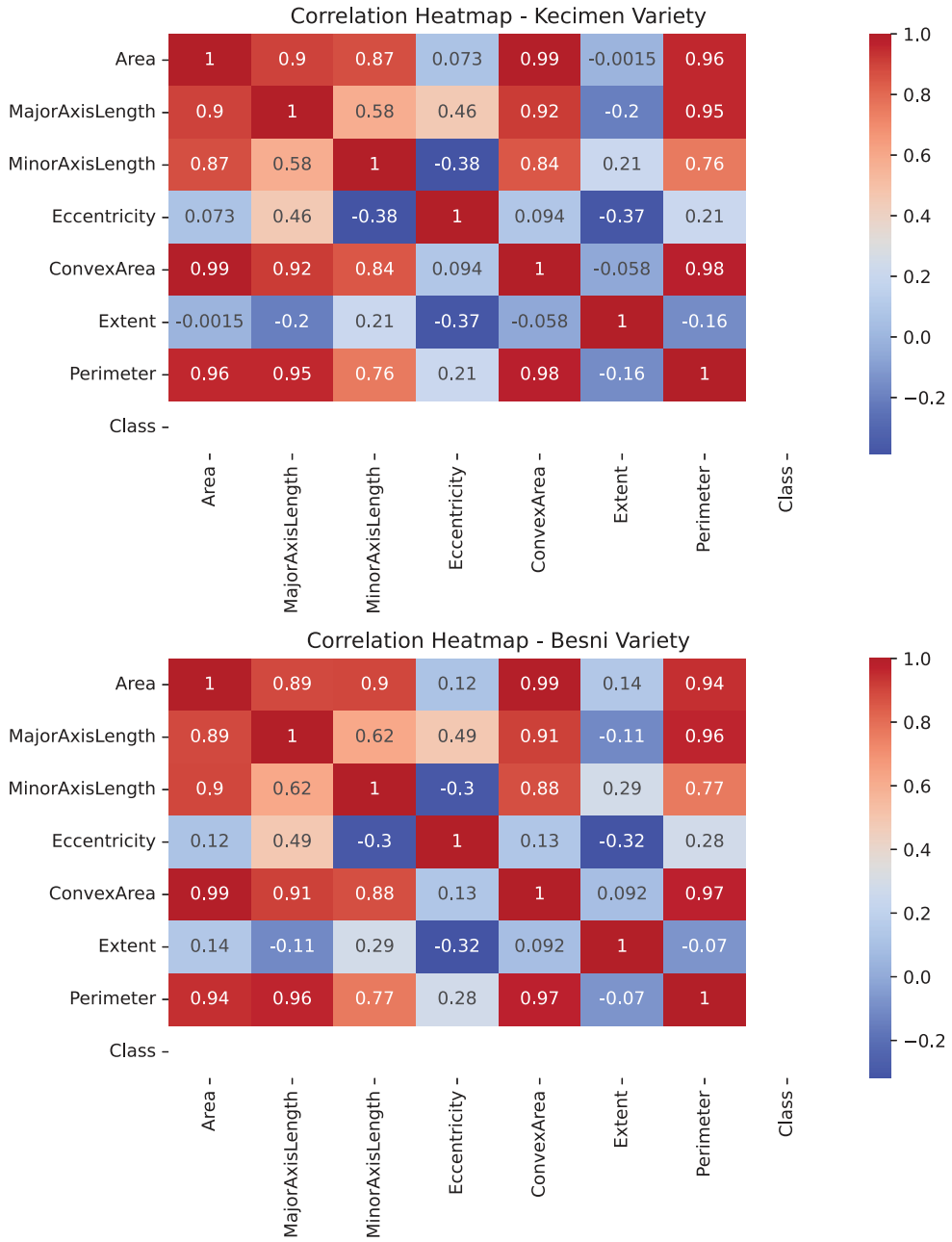

Figure 5.3 A pair of correlation heat maps, one for each variety of raisin

The correlation heat maps show that we might have a multicollinearity issue; that's because there are strong correlations between several pairs of variables. We'll

demonstrate how to test for multicollinearity and discuss how to handle it during model development.

5.2.2 Fitting the model

We'll fit a multiple logistic regression by calling the `logit()` method from Python's `statsmodels` library; `Class` will be our dependent variable, whereas `Area`, `MajorAxisLength`, `MinorAxisLength`, `Eccentricity`, `ConvexArea`, `Extent`, and `Perimeter` will be our independent variables. We are regressing `Class` against all the morphological features in the `raisins` data frame to predict the variety of raisin.

We import the `statsmodels` library and then fit our regression, `log_model`:

- `smf.logit(…)` specifies the logistic regression and assigns `Class` as the dependent variable and the remaining variables as predictors.
- `fit()` fits the initialized model to the `raisins` data frame and estimates the parameters of the regression using the specified data.
- `summary()` returns a regression table that contains the coefficient estimates, p-values, and goodness of fit statistics.

Here's the code and the model output:

```
>>> import statsmodels.formula.api as smf
>>> log_model = smf.logit('Class ~ Area + '
>>>                       'MajorAxisLength + '
>>>                       'MinorAxisLength + '
>>>                       'Eccentricity + '
>>>                       'ConvexArea + '
>>>                       'Extent + '
>>>                       'Perimeter', data = raisins).fit()
>>> print(log_model.summary())
Optimization terminated successfully.
         Current function value: 0.338384
         Iterations 9
                           Logit Regression Results
==============================================================================
Dep. Variable:                  Class   No. Observations:                  900
Model:                          Logit   Df Residuals:                      892
Method:                           MLE   Df Model:                            7
Date:                Wed, 27 Mar 2024   Pseudo R-squ.:                  0.5118
Time:                        15:03:56   Log-Likelihood:                -304.55
converged:                       True   LL-Null:                       -623.83
Covariance Type:            nonrobust   LLR p-value:                1.196e-133
==============================================================================

                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -2.6642      7.028     -0.379      0.705     -16.439      11.111
Area           0.0005      0.000      4.027      0.000       0.000       0.001
MajorAxisLength
              -0.0447      0.016     -2.801      0.005      -0.076      -0.013
MinorAxisLength
              -0.0899      0.027     -3.342      0.001      -0.143      -0.037
```

```
Eccentricity
           -3.5248      4.909      -0.718       0.473     -13.147       6.097
ConvexArea
           -0.0004      0.000      -3.446       0.001      -0.001      -0.000
Extent     -0.6051      2.697      -0.224       0.822      -5.891       4.681
Perimeter   0.0361      0.007       5.464       0.000       0.023       0.049
```

The null hypothesis for a logistic regression is that the predictors have no effect on distinguishing one class from another. So, if the p-value for the model is greater than the 5% threshold for significance, we will fail to reject the null hypothesis. But if the p-value is less than 5%, we should instead reject the null hypothesis and conclude that the morphological features of raisins can be used to detect one variety from the other. The log-likelihood ratio (LLR) p-value, which more or less equals the p-value in the F-statistic from a linear regression, is equal to 0; therefore, the null hypothesis should be rejected.

However, not every predictor is statistically significant; `Eccentricity` has a p-value equal to `0.473`, and `Extent` has a p-value equal to `0.822`. Rather than further evaluating this model, we will instead fit a reduced model that excludes, at a minimum, those two variables. But first, we'll demonstrate how to test for multicollinearity, evaluate the results, and weigh alternative courses of action. The easiest way of treating multicollinearity is to discard the offending variables. So, we may decide to reduce our next regression by more than two predictors. (The intercept's p-value is `0.705`, indicating that it is also not statistically significant—but because intercepts are generally less critical for interpretation and prediction, we typically don't exclude them.)

Multicollinearity in regression analysis is measured by a statistic called the variance inflation factor (VIF). It assesses how much the variance of an estimated regression coefficient is increased—or inflated—due to multicollinearity among the predictors. Multicollinearity is present when at least one of the predictors has a VIF equal to or greater than 1. However, some form of remediation is required only when one or more of the predictors has a VIF equal to or greater than 10.

The formula to calculate the VIF for a single predictor X_j is

$$\mathrm{VIF}_j = \frac{1}{1 - R_j^2}$$

We get R^2 by fitting a linear regression in which one of the predictors is regressed against the remaining predictors. For instance, to get the VIF for `MinorAxisLength`, we would regress that variable against the other predictors in our multiple logistic model, get the R^2 statistic, and then plug that value into the VIF formula. Let's demonstrate.

We first define our dependent variable, independent variable, and constant:

```
>>> y = raisins['MinorAxisLength']

>>> x = raisins[['Area', 'MajorAxisLength', 'ConvexArea', 'Extent', \
             'Eccentricity', 'Perimeter']]

>>> x = sm.add_constant(x)
```

Then we fit our linear regression by making a call to the `OLS()` method from the `statsmodels` library:

```
>>> import statsmodels.api as sm
>>> multicollinearity_test = sm.OLS(y, x).fit()
```

And finally, rather than instruct Python to return a regression table, we instead write two short lines of code to get just the R^2 statistic:

```
>>> r2 = multicollinearity_test.rsquared
>>> print(r2)
0.9751270231122477
```

So, the VIF for `MinorAxisLength` is

$$\text{VIF}_{\text{MinorAxisLength}} = \frac{1}{1 - 0.97513}$$

This comes to 40.21.

Fortunately, it's not necessary to repeat this exercise by fitting a linear model for every predictor in our multiple logistic regression. The following snippet of code automatically returns the VIF for every predictor:

```
>>> from patsy import dmatrices
>>> from statsmodels.stats.outliers_influence
>>> import variance_inflation_factor

>>> y, X = dmatrices('Class ~ Area + '
>>>                  'MajorAxisLength + '
>>>                  'MinorAxisLength + '
>>>                  'ConvexArea + '
>>>                  'Extent + '
>>>                  'Eccentricity + '
>>>                  'Perimeter',
>>>                  data = raisins,
>>>                  return_type = 'dataframe')

>>> vif_df = pd.DataFrame()
>>> vif_df['variable'] = X.columns

>>> vif_df['VIF'] = [variance_inflation_factor(X.values, i) \
>>>                  for i in range(X.shape[1])]

>>> print(vif_df)
         variable          VIF
0       Intercept  1117.154039
1            Area   404.773424
2  MajorAxisLength   129.289901
3  MinorAxisLength    40.204275
4      ConvexArea   446.016527
```

First of two library and module combinations required to execute subsequent lines of code

Second of two library and module combinations required to execute subsequent lines of code

Creates design matrices for the dependent and independent variables

Data is pulled from the specified data source and returned as a data frame.

Initializes an empty data frame

Adds a column to the data frame and populates it with the predictor variable names

Calculates the VIF for each predictor and stores the results in the data frame

Prints the data frame, which contains the VIFs for each predictor

```
5           Extent     1.600679
6     Eccentricity     5.194067
7        Perimeter   184.388548
```

So, as we suspected might be the case once we examined the correlation heat maps, we have a serious multicollinearity issue. Only the statistically insignificant predictors `Extent` and `Eccentricity`, have VIFs below 10.

Multicollinearity can be remediated by combining predictors or by fitting a different type of model to the data. However, the most common method is to simply fit a reduced model. We previously decided to fit a second logistic regression without `Extent` and `Eccentricity` because, according to our first regression, neither of those predictors has a statistically significant effect on detecting one variety of raisin versus the other. We will next regress `Class` against `MinorAxisLength` and `Perimeter` only—although this doesn't entirely resolve our multicollinearity issue, it mitigates the effect and, at the same time, furthers our commitment to fit and evaluate a multiple logistic regression.

So, once more, we make a call to the `logit()` method:

```
>>> reduced_model = smf.logit('Class ~ MinorAxisLength + Perimeter', \
>>>                           data = raisins).fit()
>>> print(reduced_model.summary())
Optimization terminated successfully.
         Current function value: 0.346417
         Iterations 8
                           Logit Regression Results
==============================================================================
Dep. Variable:                  Class   No. Observations:                  900
Model:                          Logit   Df Residuals:                      897
Method:                           MLE   Df Model:                            2
Date:                Thu, 28 Mar 2024   Pseudo R-squ.:                 0.5002
Time:                        19:06:19   Log-Likelihood:                -311.78
converged:                       True   LL-Null:                       -623.83
Covariance Type:            nonrobust   LLR p-value:                2.987e-136
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept
    -11.8048      0.878    -13.442      0.000     -13.526     -10.084
MinorAxisLength
     -0.0244      0.004     -5.586      0.000      -0.033      -0.016
Perimeter
      0.0159      0.001     13.333      0.000       0.014       0.018
==============================================================================
```

Now we have a model that is worthy of a full evaluation.

5.2.3 Interpreting and evaluating the results

Our intent here is to pull the key measures from the regression table and demonstrate how they should be interpreted and evaluated. In this section, we will provide instructions on deriving, evaluating, and even visualizing classification metrics that are used to measure the performance of a regression in predicting binary outcomes.

PSEUDO R-SQUARED

In logistic regression, pseudo R^2 is a measure used to assess the model's goodness of fit. Unlike in linear regression, where R^2 represents the proportion of variance explained by the predictors, pseudo R^2 represents the model's ability to explain variation in the binary dependent variable.

There are actually several pseudo R^2 measures common to logistic regression; the statsmodels library returns what's called McFadden's pseudo R^2, which is the most popular. McFadden's R^2 is equal to 1 minus the quotient between the log-likelihood of the model and the log-likelihood of a model with just an intercept term and no predictors:

$$\text{McFadden's } R^2 = 1 - \frac{L_{\text{model}}}{L_{\text{null}}}$$

It therefore measures the added value of the independent variables in predicting binary outcomes; so, the higher McFadden's R^2, the better the model fits the data. The log-likelihoods are stored as attributes in the fitted model object and are printed to the regression table.

This means

$$\text{McFadden's } R^2 = 1 - \frac{-311.78}{-623.83} = 0.5002$$

Log-likelihood is a measure of how well a statistical model, including logistic regression, fits the observed data. Specifically, it represents the logarithm of the likelihood function, which is the probability of observing the data given the parameters of the model. The log-likelihood of the null model is a baseline of sorts, and the log-likelihood of the model is the predictive value of the independent variables above that baseline. Therefore, the greater the difference between the log-likelihoods of the model and the null model, the higher McFadden's R^2, and the better the model fits the data.

Log-likelihoods are difficult to interpret in isolation, but they can readily be compared to like measures from competing models fitted to the same data. Therefore, McFadden's R^2, which of course is derived from the log-likelihoods, is also a valuable metric for the purposes of model comparison. But unlike the log-likelihoods, McFadden's R^2 is easy to evaluate and interpret against the fitted model. It will always equal some number between 0 and 1, where values as "low" as 0.40 correspond to an excellent fit. Thus, a McFadden's R^2 equal to 0.50 suggests that `MinorAxisLength` and `Perimeter` are accurate predictors for distinguishing one variety of raisin from the other.

LLR P-VALUE AND P>|Z|

Because the reduced model excluded statistically insignificant predictors from the full model and other predictors with high multicollinearity, it's not surprising that we again get a p-value equal to 0. We've simplified the model without making a sacrifice in statistical significance. So, regressing `Class` against just `MinorAxisLength` and

`Perimeter` leads us to the same conclusion as did regressing `Class` against the full complement of morphological features: because the p-value for the model is less than 5%, we should again reject the null hypothesis and now conclude that `MinorAxisLength` and `Perimeter`, which both have p-values also equal to 0, are useful for detecting one variety of raisin versus the other.

COEF

The intercept (β_0) equals `-11.8048`, the coefficient estimate for `MinorAxisLength` (β_1) is equal to `-0.0244`, and the same for `Perimeter` (β_2) is equal to `0.0159`. We therefore get the fitted regression by plugging these three values into the logistic equation:

$$P(Y = 1|X) = \frac{1}{1 + e^{-(-11.8048 + -0.0244(X_1) + 0.0159(X_2)}}$$

or

$$P(\text{Besni}) = \frac{1}{1 + 2.72^{-(-11.8048 + -0.0244(\text{MinorAxisLength}) + 0.0159(\text{Perimeter}))}}$$

Once more, the purpose of logistic regression is to predict the probability of the dependent variable being equal to 1. And because we previously coded Kecimen raisins as 0 and Besni raisins as 1, we are therefore predicting the probability of the raisin variety equaling Besni.

Let's create a subset of the `raisins` data frame that contains just the variables `MinorAxisLength`, `Perimeter`, and `Class`:

```
>>> raisins_subset = raisins[['MinorAxisLength', 'Perimeter', 'Class']]
```

After all, only these three variables from our original data frame now factor into our analysis, so it makes sense to subset the data by removing unnecessary variables and thus retain only what is needed.

Next we create a new `raisins_subset` variable called `probability` that corresponds to the coefficient estimates and their place in the logistic equation. This operation first requires that we import the `math` library:

```
>>> import math
>>> raisins_subset['probability'] = 1 / (1 + 2.72**(-(-11.8048 +
>>>                (-.0244 * raisins_subset['MinorAxisLength']) +
>>>                (.0159 * raisins_subset['Perimeter']))))
```

Let's get a glimpse—two glimpses, actually—of the `raisins_subset` data frame by making successive calls to the `head()` and `tail()` methods:

```
>>> print(raisins_subset.head(n = 3))
   MinorAxisLength  Perimeter  Class  probability
0           253.29    1184.04      0     0.698821
1           243.03    1121.79      0     0.525426
2           266.33    1208.58      0     0.713766
```

```
>>> print(raisins_subset.tail(n = 3))
     MinorAxisLength  Perimeter  Class  probability
897           298.84    1292.83      1     0.811597
898           254.18    1258.55      1     0.881365
899           215.27    1272.86      1     0.960193
```

Take a look at the first and last observations in the data. When `MinorAxisLength` equals `253.29` and `Perimeter` equals `1184.04`, our logistic regression gives a 70% probability that `Class` equals `1`; when `MinorAxisLength` equals `215.27` and `Perimeter` equals `1272.86`, our model gives a 96% probability of `Class` equaling `1`.

This was just an interim step, however. We now need to mutate the values in `probability` to binary predictions that coincide with the observed outcomes. The following snippet of code mutates the values in `probability` to either `0` or `1` and stores the results in a new variable called `prediction`: `0` if the `probability` is less than 0.50 or `1` otherwise. This operation requires the `pandas` and `numpy` libraries. Although 0.5 is a common default cutoff—meaning we predict a 1 when the probability is at least 50%—this threshold can be adjusted depending on the context. For instance, in situations where false positives are more costly than false negatives (or vice versa), a higher or lower cutoff might be more appropriate to optimize the model's performance:

```
>>> import numpy as np
>>> raisins_subset['prediction'] = (raisins_subset['probability'] \
>>>                                 .apply(lambda x: 0 if x < 0.5 else 1))
```

Successive calls to the `head()` and `tail()` methods display the top three and bottom three observations in the `raisins_subset` data frame:

```
>>> print(raisins_subset.head(n = 3))
   MinorAxisLength  Perimeter  Class  probability  prediction
0           253.29    1184.04      0     0.698821           1
1           243.03    1121.79      0     0.525426           1
2           266.33    1208.58      0     0.713766           1

>>> print(raisins_subset.tail(n = 3))
     MinorAxisLength  Perimeter  Class  probability  prediction
897           298.84    1292.83      1     0.811597           1
898           254.18    1258.55      1     0.881365           1
899           215.27    1272.86      1     0.960193           1
```

Now that we have a binary outcome variable and a binary prediction variable, we can calculate a series of classification metrics to further evaluate the predictive power of our logistic regression.

5.2.4 Calculating and evaluating classification metrics

Classification metrics are used to evaluate the performance of a binary classification model, including logistic regression. These metrics provide insights into how well, or not so well, the model is able to classify instances into their correct classes.

CONFUSION MATRIX

A *confusion matrix* is a table that is typically used to describe the performance of a classification model like logistic regression. It neatly visualizes the performance of an algorithm by comparing actual classes (from our data set) with predicted classes. The matrix consists of four main components: true positives (*tp*), false positives (*fp*), true negatives (*tn*), and false negatives (*fn*).

These components represent the counts of correct and incorrect predictions made by the model. By analyzing the confusion matrix, we can intelligently and quantitatively evaluate the performance of our logistic regression. The confusion matrix with respect to our classification of Turkish raisins is shown in table 5.1.

Table 5.1 Confusion matrix for raisin classification. A confusion matrix stores classification metrics from a regression with actual and predicted binary outcomes.

		Predicted	
		Kecimen (0)	Besni (1)
Actual	Kecimen (0)	*tn*	*fp*
	Besni (1)	*fn*	*tp*

The following snippet of code returns the number of records in the `raisins_subset` data frame where `Class` and `prediction` both equal `0`; this is therefore the count of true negatives, or the number of instances where `Class` equals `0` and was then predicted to equal `0`. The `len()` method typically returns the number of items in an object:

```
>>> count = len(raisins_subset[(raisins_subset['Class'] == 0) &
>>>                            (raisins_subset['prediction'] == 0)])
>>> print(count)
397
```

So, out of the 450 observations where the variety of raisin equals Kecimen, our model correctly predicted the outcome 397 times.

The next snippet of code returns the inverse of that, or the number of instances where Kecimen raisins were incorrectly predicted to be of the Besni variety. This is the count of false positives:

```
>>> count = len(raisins_subset[(raisins_subset['Class'] == 0) &
>>>                            (raisins_subset['prediction'] == 1)])
>>> print(count)
53
```

The number of observations where `Class` equals `1` and `prediction` equals `1` represents the number of times the model correctly predicted the variety of raisin to equal Besni, which is the count of true positives:

```
>>> count = len(raisins_subset[(raisins_subset['Class'] == 1) &
>>>                            (raisins_subset['prediction'] == 1)])
print(count)
384
```

Thus, from the 450 observations where the variety of raisin equals Besni, our logistic regression correctly predicted that outcome 384 times.

And finally, we get the inverse of the true positives, the false negatives, by counting the number of instances where `Class` equals `1` but `prediction` equals `0`:

```
>>> count = len(raisins_subset[(raisins_subset['Class'] == 1) &
>>>                            (raisins_subset['prediction'] == 0)])
>>> print(count)
66
```

We get a populated confusion matrix by passing the `raisins_subset` variables `Class` and `prediction` to the `pd.crosstab()` method:

```
>>> confusion_matrix = pd.crosstab(raisins_subset['Class'], \
>>>                                raisins_subset['prediction'])
>>> print(confusion_matrix)
prediction    0    1
Class
0           397   53
66  384
```

Thus:

$$\begin{aligned} tn &= 397 \\ fp &= 53 \\ tp &= 384 \\ fn &= 66 \\ n &= 900 \end{aligned}$$

where

- tn equals the number of true negatives.
- fp equals the number of false positives.
- tp equals the number of true positives.
- tn equals the number of false negatives.
- n equals the total record count, which, again, is evenly divided between Kecimen and Besni raisins; it also equals the total number of predictions.

We'll plug these values into other formulas to get additional classification metrics that help us further evaluate the predictive power of our model.

SENSITIVITY

Sensitivity is the true positive rate (tpr); it equals the number of true positives divided by the sum of true positives and false negatives. The quotient is then multiplied by 100 to get a percentage:

$$\text{sensitivity} = \left(\frac{tp}{tp + fn}\right) 100$$

or

$$\text{sensitivity} = \left(\frac{384}{384 + 66}\right) 100$$

There are two ways to get the true positive rate. We can use Python as a calculator:

```
>>> sensitivity = (tp / (tp + fn)) * 100
>>> print(sensitivity)
85.33333333333334
```

Or we can pass the `raisins_subset` variables `Class` and `prediction` to the `recall_score()` method from the scikit-learn (`sklearn`) library:

```
>>> from sklearn.metrics import recall_score
>>> tpr = recall_score(raisins_subset['Class'], \
>>>                    raisins_subset['prediction'])
>>> print(tpr * 100)
85.33333333333334
```

Either way, we get a true positive rate of 85.33%. In other words, our logistic regression correctly identified positive instances 85% of the time, which is pretty good.

SPECIFICITY

Specificity is the true negative rate (tnr); it equals the number of true negatives divided by the sum of true negatives and false positives. The quotient is then transformed to a percentage by multiplying it by 100:

$$\text{specificity} = \left(\frac{tn}{tn + fp}\right) 100$$

or

$$\text{specificity} = \left(\frac{397}{397 + 53}\right) 100$$

Once more, we can get the true negative rate by performing another simple arithmetic operation:

```
>>> specificity = (tn / (tn + fp)) * 100
>>> print(specificity)
88.22222222222223
```

Or we can again call the `recall_score` method(). In addition to passing the variables `Class` and `prediction`, we must also pass a third argument this time, `pos_label = 0`, which instructs `recall_score()` to fix its computations on the negative class:

```
>>> tnr = recall_score(raisins_subset['Class'], \
>>>                    raisins_subset['prediction'], pos_label = 0)
>>> print(tnr * 100)
88.22222222222223
```

So, the true negative rate, or the proportion of negative instances correctly predicted by our logistic regression, is 88.22%, which is also good. Thus, our model performs (almost) equally well across classes.

FALSE POSITIVE RATE

The false positive rate (*fpr*) measures the proportion of actual negatives incorrectly classified as positives by the logistic regression. It equals the number of false positives divided by the sum of false positives and true negatives, which is then multiplied by 100 to get a percentage:

$$tnr = \left(\frac{fp}{fp + tn}\right) 100$$

or

$$tnr = \left(\frac{53}{53 + 397}\right) 100$$

There's no function available to get the false positive rate, but of course, we can again use Python as a calculator to determine it. It is the inverse of the true negative rate:

```
>>> false_positive_rate = (fp / (fp + tn)) * 100
>>> print(false_positive_rate)
11.777777777777777
```

Our model incorrectly classified the negative instances less than 12% of the time.

FALSE NEGATIVE RATE

The false negative rate (*fnr*) measures the proportion of actual positives incorrectly classified as negatives by our model. We get the false negative rate by dividing the number of false negatives by the sum of false negatives and true positives, and then, of course, multiplying that result by 100 to get a percentage of records that meet this criterion:

$$fnr = \left(\frac{fn}{fn + tp}\right) 100$$

or

$$fnr = \left(\frac{66}{66 + 384}\right) 100$$

We can get the result in Python just as before. It is the inverse of the true positive rate:

```
>>> false_negative_rate = (fn / (tp + fn)) * 100
>>> print(false_negative_rate)
14.666666666666666
```

So, our model missed on just 15% of the actual positives.

False positive rates vs. false negative rates

Depending on the scenario, false positives and false negatives are not equally severe. Consider, for instance, airport security screening. False positives, or false alarms, are the result of incorrectly identifying harmless objects as potential threats, which of course inconveniences passengers and creates potential delays. But those consequences are not nearly as severe as what might occur, such as a security breach or even a terrorist attack, if the false negatives were high.

Or consider medical diagnoses for life-threatening diseases. Incorrectly flagging healthy individuals for medical care is not as consequential as missing the true positives, which result in undiagnosed cases and delays in treatment.

We can also consider the incarceration of innocent individuals. These false positives are absolutely more severe than providing freedom to those who are actually guilty.

ACCURACY

Model *accuracy* is the ratio of correct predictions to the total number of predictions. Mathematically, it is the sum of true positives and true negatives divided by the number of records in the data (assuming that a prediction was applied to each observation), multiplied by 100:

$$\text{accuracy} = \left(\frac{tp + tn}{n}\right) 100$$

or

$$\text{accuracy} = \left(\frac{384 + 397}{900}\right) 100$$

We can naturally get the same result by performing yet another arithmetic operation in Python:

```
>>> accuracy = ((tp + tn) / n) * 100
>>> print(accuracy)
86.77777777777777
```

Of course, the closer the model accuracy is to 100%, the better. Our logistic regression isn't quite there, but an 87% accuracy rate is fairly impressive.

MISCLASSIFICATION RATE

The *misclassification rate* is the opposite, or inverse, of model accuracy. It is the proportion of false predictions to the record count, multiplied by 100:

$$\text{misclassification rate} = \left(\frac{fp + fn}{n}\right) 100$$

or

$$\text{misclassification rate} = \left(\frac{53 + 66}{900}\right) 100$$

This should be approximately 13%:

```
>>> misclassification_rate = ((fp + fn) / n) * 100
>>> print(misclassification_rate)
13.222222222222221
```

So, our model falsely, or incorrectly, predicted the variety of raisin just 13% of the time.

AREA UNDER THE ROC CURVE

The *area under the ROC curve* (AUC) is a measure that quantifies the overall performance of a classification problem, including logistic regression, across all possible classification thresholds. The `roc_auc_score()` method from the `sklearn` library compares the predictions against the true binary labels and returns a statistic between 0 and 1.

The AUC may or may not align with the model accuracy rate. Whereas model accuracy calculates the ratio of correct predictions to the total number of predictions, AUC measures the ability of the model to effectively discriminate between positive and negative classes. The difference might seem subtle, but consider an imbalanced data set where one class is predominant over the other. For example, consider a data set where 90 students pass an exam and 10 fail: accuracy may be high if the model predicts the dominant class correctly most of the time, but the AUC may be lower than the accuracy rate if the model struggles to effectively distinguish between the classes.

When the data is evenly divided between classes (remember, we have 450 Kecimen raisins and 450 Besni raisins in our data), and if the model performs equally well across classes, the AUC statistic should align well with model accuracy:

```
>>> from sklearn.metrics import roc_auc_score
>>> auc = roc_auc_score(raisins_subset['Class'], \
>>>                     raisins_subset['prediction'])
>>> print(auc)
0.8677777777777779
```

And it does. In fact, the AUC and the model accuracy rate are exactly the same.

In general, an AUC close to 1 indicates very good model performance, thereby suggesting that our logistic regression has high discriminatory power; if the AUC were instead close to 0.5, it would suggest that model performance was no better than random guessing. But more specifically, what qualifies as a "good" AUC statistic depends to a large degree on the context and the problem at hand. When the problem is innocuous, like distinguishing one variety of raisin from the other, an AUC equal to 87% is considered excellent; but if one variety happened to be poisonous, 87% might be unsatisfactory.

ROC CURVE

A *receiver operating characteristic* (ROC) curve is a graphical representation of the AUC. It plots the false positive rate along the *x* axis and the true positive rate along the *y* axis. It illustrates the potential trade-off between the false and true positive rates across different threshold values. A ROC curve that hugs the upper-left quadrant of the plot indicates strong model performance whereby the true positives are maximized while the true negatives are minimized (see figure 5.4); this increases the area under it, driving the AUC statistic closer to 1.

Here's the snippet of Python code:

```
>>> from sklearn.metrics import roc_curve
>>> (fpr, tpr,
>>>  thresholds = roc_curve(raisins_subset['Class'],
>>>                         raisins_subset['prediction'])
>>> plt.plot(fpr, tpr, color = 'blue',
>>>          lw = 2, label = 'ROC curve')
>>> plt.plot([0, 1], [0, 1], color = 'gray',
>>>          linestyle = '--')
>>> plt.title('ROC Curve\nAUC = 0.87')
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
>>> plt.legend(loc = 'lower right')
>>> plt.grid()
>>> plt.show()
```

Imports the required library and module

Calculates the false positive and true positive rates

Initializes the plot and draws a blue solid line to represent the ROC curve

Draws a perfectly diagonal dashed line that represents an AUC equal to 0.50

Sets the title

Sets the x-axis label

Sets the y-axis label

Adds a legend in the lower-right quadrant of the plot

Adds vertical and horizontal grid lines

Displays the plot

Figure 5.4 shows the plot.

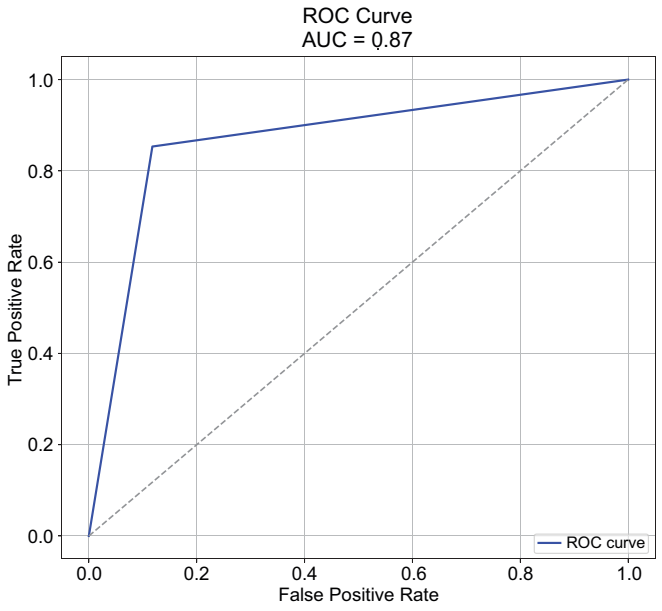

Figure 5.4 A ROC curve. The false positive rate is plotted along the *x* axis, and the true positive rate is plotted along the *y* axis. When the curve, represented by the solid line, hugs the upper-left quadrant of the plot, as it does here, it increases the area under the curve. The dashed line represents an AUC equal to 0.5, or the equivalent of random guessing.

We learned how logistic regression can effectively solve a real-world classification problem, but the journey to get there was a bit of a crooked path. Our preliminary analysis of the data showed that a full model—where the binary dependent variable is regressed against every other variable in the data—might cause a multicollinearity issue. We demonstrated how to fit a multiple logistic regression and then how to test for multicollinearity, which led us to fit a reduced model where the same dependent variable was then regressed against a subset of our original predictors. We showed how to interpret and evaluate goodness of fit measures and other model outputs, how to apply the coefficient estimates with some data wrangling to make predictions, and how to compute classification metrics that are typically used to measure model performance. In the next chapter, we will solve another classification problem, but with very different modeling techniques.

Summary

- Logistic regression is a statistical method used to model the relationship between a binary outcome variable and one or more predictors. Unlike linear regression, which predicts continuous outcomes, logistic regression predicts the probability that an observation belongs to one class versus the other. Logistic regression applies the logistic function, or sigmoid function, to transform the linear combination of predictors into probabilities, which are then used to make binary predictions. Logistic regression models are regularly fit in many fields, including medicine, finance, marketing, and the social sciences.
- Goodness of fit should be evaluated from at least a trio of measures: McFadden's pseudo R^2, which represents a model's ability to explain variation in the binary dependent variable above and beyond a null model that contains just an intercept term; the LLR p-value, which measures a model's statistical significance; and the p-values for the individual predictors, which indicate whether the relationship between a predictor and the binary outcome variable is statistically significant.
- Classification metrics—true and false positive rates, true and false negative rates, accuracy and misclassification rates, and the area under the receiver operating characteristic curve—are essential for evaluating the performance of a logistic regression and its ability to accurately predict binary outcomes. By analyzing these metrics, you can draw conclusions about a model's strengths and weaknesses, identify opportunities for improvement, and make informed decisions about model optimization and deployment.
- Data exploration, which typically involves a mix of computing basic statistics and creating graphical displays of the data, should scale with the length and width of the data set. The value proposition of data exploration, when done correctly and thoroughly, lies in its ability to identify patterns, outliers, and potential issues with the data that could drive variable selection and other model development decisions.