APPENDIX

CHAPTER 1: INTRODUCTION TO DATA SCIENCE IN PYTHON

ACTIVITY 1.01: TRAIN A SPAM DETECTOR ALGORITHM

SOLUTION

1. Open a new Colab notebook.
2. Import **pandas**:

```
import pandas as pd
```

3. Create a variable containing the URL to the dataset:

```
file_url = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter01/Dataset'\
           '/dataset_44_spambase.csv'
```

4. Load the dataset using the **.read_csv()** method from **pandas** into a variable called **df**:

```
df = pd.read_csv(file_url)
```

5. Print the **df** DataFrame:

```
print(df)
```

You should get the following output:

```
      word_freq_make  word_freq_address  ...  capital_run_length_total  class
0               0.00               0.64  ...                       278      1
1               0.21               0.28  ...                      1028      1
2               0.06               0.00  ...                      2259      1
3               0.00               0.00  ...                       191      1
4               0.00               0.00  ...                       191      1
...              ...                ...  ...                       ...    ...
4596            0.31               0.00  ...                        88      0
4597            0.00               0.00  ...                        14      0
4598            0.30               0.00  ...                       118      0
4599            0.96               0.00  ...                        78      0
4600            0.00               0.00  ...                        40      0

[4601 rows x 58 columns]
```

Figure 1.53: Output of df

The preceding output shows some rows and columns of the loaded DataFrame.

6. Extract the **`class`** target variable using the **`.pop()`** method from **`pandas`** and save the result in a variable called **`target`**:

```
target = df.pop('class')
```

7. Print the **`target`** variable.

```
print(target)
```

You should get the following output:

```
0        1
1        1
2        1
3        1
4        1
        ..
4596     0
4597     0
4598     0
4599     0
4600     0
Name: class, Length: 4601, dtype: int64
```

Figure 1.54: The target variable

8. Import the **`RandomForestClassifier`** class from **`sklearn.ensemble`**:

```
from sklearn.ensemble import RandomForestClassifier
```

9. Create a new variable called **`seed`**, which will take the value **`168`**. This is used for getting reproducible outputs:

```
seed = 168
```

10. Instantiate **`RandomForestClassifier`** with the **`random_state=seed`** parameter and save it into a variable called **`rf_model`**:

```
rf_model = RandomForestClassifier(random_state=seed)
```

11. Train the model with the **`.fit()`** method with **`df`** and **`target`** as parameters:

```
rf_model.fit(df, target)
```

The output will be as follows:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=168,
                       verbose=0, warm_start=False)
```

Figure 1.55: RandomForest logs

12. Make predictions with the trained model using the **`.predict()`** method and **`df`** as a parameter and save the results into a variable called **`preds`**:

```
preds = rf_model.predict(df)
```

13. Print the **`preds`** variable.

```
print(preds)
```

You should get the following output:

```
[1 1 1 ... 0 0 0]
```

Figure 1.56: RandomForest predictions

The preceding output shows the model predictions for the DataFrame.

14. Import the **`accuracy_score`** method from **`sklearn.metrics`**:

```
from sklearn.metrics import accuracy_score
```

15. Calculate **`accuracy_score()`** with **`target`** and **`preds`** as parameters and save the results in a variable called **`acc_score`**:

```
acc_score = accuracy_score(target, preds)
```

16. Print the **`acc_score`** variable:

```
print(acc_score)
```

You should get the following output:

```
0.9958704629428385
```

Figure 1.57: Accuracy score

From the preceding output, we notice that we achieved an accuracy score of 99.59. So, the model is making the right prediction in 99.59% of the cases.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3iWWhjz.
>
> You can also run this example online at https://packt.live/2Fyk5fc.

CHAPTER 2: REGRESSION

ACTIVITY 2.01: FITTING A LOG-LINEAR MODEL USING THE STATSMODELS FORMULA API

SOLUTION

1. Open a new Colab notebook file.
2. Load the necessary Python modules by entering the following code snippet into a single Colab notebook cell. Press the **Shift** and **Enter** keys together to run the block of code:

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
from sklearn.model_selection import train_test_split
```

The lines of code that follow use the **`import`** keyword to load various Python modules into our programming environment.

3. Next, load the **`Boston.CSV`** file and assign the variable name **`rawBostonData`** to it by running the following code:

```
rawBostonData = pd.read_csv\
                ('https://raw.githubusercontent.com'\
                 '/PacktWorkshops/The-Data-Science-Workshop'\
                 '/master/Chapter02/Dataset/Boston.csv')
```

4. Check for missing values (null values) in the data frame and then drop them to have a clean dataset:

```
rawBostonData = rawBostonData.dropna()
```

5. Check for duplicate records in the data frame and drop them to have a clean dataset:

```
rawBostonData = rawBostonData.drop_duplicates()
```

6. Rename the data frame columns so that they are meaningful. Be mindful to match the column names exactly since leaving out even white spaces in the name strings will result in an error. For example, this string, **`ZN`**, has a white space before and after and it is different from **`ZN`**. After renaming, print the head of the new data frame:

```
renamedBostonData = rawBostonData.rename\
                    (columns = {'CRIM':'crimeRatePerCapita',\
                                ' ZN ':'landOver25K_sqft',\
                                'INDUS ':'non-retailLandProptn',\
                                'CHAS':'riverDummy',\
                                'NOX':'nitrixOxide_pp10m',\
                                'RM':'AvgNo.RoomsPerDwelling',\
                                'AGE':'ProptnOwnerOccupied',\
                                'DIS':'weightedDist',\
                                'RAD':'radialHighwaysAccess',\
                                'TAX':'propTaxRate_per10K',\
                                'PTRATIO':'pupilTeacherRatio',\
                                'LSTAT':'pctLowerStatus',\
                                'MEDV':'medianValue_Ks'})
```

7. Divide the data frame into train and test sets, as shown in the following code snippet:

```
X = renamedBostonData.drop('crimeRatePerCapita', axis = 1)
y = renamedBostonData[['crimeRatePerCapita']]
seed = 10
test_data_size = 0.3
X_train, X_test, y_train, y_test = train_test_split\
                                   (X, y, \
                                    test_size = test_data_size,\
                                    random_state = seed)
train_data = pd.concat([X_train, y_train], axis = 1)
test_data = pd.concat([X_test, y_test], axis = 1)
```

We choose a test data size of 30%, which is **`0.3`**. The **`train_test_split`** function is used to achieve this. We set the seed of the random number generator so that we can obtain a reproducible split each time we run this code. An arbitrary value of **`10`** is used here. It is good model building practice to divide a dataset being used to develop a model into at least two parts. One part is used to develop the model and it is called training set (**`X_train`** and **`y_train`** combined).

8. Now, define a linear regression model and assign it to a variable.

The NumPy **`log`** function is used to transform the dependent variable (**`crimeRatePerCapita`**) in the formula string:

```
logLinearModel = smf.ols\
                 (formula = \
                  'np.log(crimeRatePerCapita) ~ medianValue_Ks',\
                  data=train_data)
```

9. Call the fit method of the model instance and assign the results of the method to a variable:

```
logLinearModResult = logLinearModel.fit()
```

10. Print a summary of the results stored in the variable created in *Step 13*:

```
print(logLinearModResult.summary())
```

You should get the following output:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:     np.log(crimeRatePerCapita)   R-squared:                       0.238
Model:                                    OLS   Adj. R-squared:                  0.236
Method:                         Least Squares   F-statistic:                     109.9
Date:                        Mon, 14 Oct 2019   Prob (F-statistic):           1.48e-22
Time:                                04:08:39   Log-Likelihood:                -727.67
No. Observations:                         354   AIC:                             1459.
Df Residuals:                             352   BIC:                             1467.
Df Model:                                   1
Covariance Type:                    nonrobust
==================================================================================
                     coef    std err          t      P>|t|      [0.025      0.975]
----------------------------------------------------------------------------------
Intercept          1.9107      0.271      7.062      0.000       1.379       2.443
medianValue_Ks    -0.1198      0.011    -10.482      0.000      -0.142      -0.097
==============================================================================
Omnibus:                       11.420   Durbin-Watson:                   1.907
Prob(Omnibus):                  0.003   Jarque-Bera (JB):               10.764
Skew:                           0.376   Prob(JB):                      0.00460
Kurtosis:                       2.594   Cond. No.                         63.7
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Figure 2.20: Expected summary of results

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/32bxZvk.
>
> You can also run this example online at https://packt.live/2QbexcA.

ACTIVITY 2.02: FITTING A MULTIPLE LOG-LINEAR REGRESSION MODEL

SOLUTION

1. Open a new Colab notebook file.
2. Load the necessary Python modules by entering the following code snippet into a single Colab notebook cell. Press the **Shift** and **Enter** keys together to run the block of code:

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
from sklearn.model_selection import train_test_split
```

The lines of code that follow use the **`import`** keyword to load various Python modules into our programming environment.

3. Next, load the **`Boston.CSV`** file and assign the variable name **`rawBostonData`** to it by running the following code:

```
rawBostonData = pd.read_csv\
                ('https://raw.githubusercontent.com'\
                 '/PacktWorkshops/The-Data-Science-Workshop'\
                 '/master/Chapter02/Dataset/Boston.csv')
```

4. Check for missing values (null values) in the data frame and drop them to have a clean dataset:

```
rawBostonData = rawBostonData.dropna()
```

5. Check for duplicate records in the data frame and drop them to have a clean dataset:

```
rawBostonData = rawBostonData.drop_duplicates()
```

6. Rename the data frame columns so they are meaningful. Be mindful to match the column names exactly as leaving out even white spaces in the name strings will result in an error. For example, this string, **ZN**, has a white space before and after and it is different from **ZN**. After renaming, print the head of the new data frame:

```
renamedBostonData = rawBostonData.rename\
                    (columns = {'CRIM':'crimeRatePerCapita',\
                                ' ZN ':'landOver25K_sqft',\
                                'INDUS ':'non-retailLandProptn',\
                                'CHAS':'riverDummy',\
                                'NOX':'nitrixOxide_pp10m',\
                                'RM':'AvgNo.RoomsPerDwelling',\
                                'AGE':'ProptnOwnerOccupied',\
                                'DIS':'weightedDist',\
                                'RAD':'radialHighwaysAccess',\
                                'TAX':'propTaxRate_per10K',\
                                'PTRATIO':'pupilTeacherRatio',\
                                'LSTAT':'pctLowerStatus',\
                                'MEDV':'medianValue_Ks'})
renamedBostonData.head()
```

The output is as follows (truncated):

	crimeRatePerCapita	landOver25K_sqft	non-retailLandProptn	riverDummy	nitrixOxide_pp10m	AvgNo.RoomsPerDwelling	ProptnOwnerOccupied
0	0.00632	18.0	2.31	0	0.538	6.575	65.2
1	0.02731	0.0	7.07	0	0.469	6.421	78.9
2	0.02729	0.0	7.07	0	0.469	7.185	61.1
3	0.03237	0.0	2.18	0	0.458	6.998	45.8
4	0.06905	0.0	2.18	0	0.458	7.147	54.2

Figure 2.21: Dataset with renamed headings

7. Divide the data frame into train and test sets, as shown in the following code snippet:

```
X = renamedBostonData.drop('crimeRatePerCapita', axis = 1)
y = renamedBostonData[['crimeRatePerCapita']]
seed = 10
test_data_size = 0.3
```

```
X_train, X_test, y_train, y_test = train_test_split\
                                   (X, y, \
                                    test_size = test_data_size,\
                                    random_state = seed)
train_data = pd.concat([X_train, y_train], axis = 1)
test_data = pd.concat([X_test, y_test], axis = 1)
```

We choose a test data size of 30%, which is **0.3**. The **train_test_split** function is used to achieve this. We set the seed of the random number generator so that we can obtain a reproducible split each time we run this code. An arbitrary value of **10** is used here. It is good model-building practice to divide a dataset being used to develop a model into at least two parts. One part is used to develop the model and it is called training set (**X_train** and **y_train** combined).

8. Define a linear regression model and assign it to a variable. Remember to use the log function to transform the dependent variable in the formula string, and also include more than one independent variable. Use ****2** to specify the interaction of order **2**:

```
multiLogLinMod = smf.ols\
                 (formula='np.log(crimeRatePerCapita) ~ \
                  (pctLowerStatus + radialHighwaysAccess \
                  + medianValue_Ks + nitrixOxide_pp10m)**2', \
                  data=train_data)
```

9. Call the **fit** method of the model instance and assign the results of the method to a new variable:

```
multiLogLinModResult = multiLogLinMod.fit()
```

10. Print a summary of the results:

```
print(multiLogLinModResult.summary())
```

The output will be as follows:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:     np.log(crimeRatePerCapita)   R-squared:                       0.884
Model:                                    OLS   Adj. R-squared:                  0.881
Method:                         Least Squares   F-statistic:                     261.5
Date:                        Fri, 17 Jul 2020   Prob (F-statistic):          7.79e-154
Time:                                10:48:45   Log-Likelihood:                -394.39
No. Observations:                         354   AIC:                             810.8
Df Residuals:                             343   BIC:                             853.3
Df Model:                                  10
Covariance Type:                    nonrobust
==============================================================================================================
                                             coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------------------------------------
Intercept                                 -5.4707      1.490     -3.671      0.000      -8.402      -2.540
pctLowerStatus                             0.1541      0.049      3.161      0.002       0.058       0.250
radialHighwaysAccess                       0.4697      0.052      9.070      0.000       0.368       0.572
medianValue_Ks                            -0.1457      0.044     -3.325      0.001      -0.232      -0.059
nitrixOxide_pp10m                          3.4509      3.000      1.150      0.251      -2.450       9.352
pctLowerStatus:radialHighwaysAccess       -0.0006      0.001     -0.576      0.565      -0.003       0.002
pctLowerStatus:medianValue_Ks             -0.0041      0.001     -4.159      0.000      -0.006      -0.002
pctLowerStatus:nitrixOxide_pp10m          -0.0783      0.081     -0.964      0.336      -0.238       0.082
radialHighwaysAccess:medianValue_Ks       -0.0027      0.001     -2.694      0.007      -0.005      -0.001
radialHighwaysAccess:nitrixOxide_pp10m    -0.4234      0.066     -6.404      0.000      -0.553      -0.293
medianValue_Ks:nitrixOxide_pp10m           0.3552      0.092      3.869      0.000       0.175       0.536
==============================================================================
Omnibus:                        4.124   Durbin-Watson:                   1.966
Prob(Omnibus):                  0.127   Jarque-Bera (JB):                4.107
Skew:                           0.175   Prob(JB):                        0.128
Kurtosis:                       3.395   Cond. No.                     3.29e+04
==============================================================================
```

Figure 2.22: A summary of a fit of the log-linear regression model of crime rate per capita that explains over 80% of the variability in the transformed dependent variable

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3l3H0j1.
>
> You can also run this example online at https://packt.live/3aIMiLL.

CHAPTER 3: BINARY CLASSIFICATION

ACTIVITY 3.01: BUSINESS HYPOTHESIS TESTING TO FIND EMPLOYMENT STATUS VERSUS PROPENSITY FOR TERM DEPOSITS

SOLUTION

1. Defining the hypothesis: With respect to employment status, let's arrive at a hypothesis that *high paying employees prefer term deposits more than other categories of employees*.

   Find the total number of customers under each employment status:

2. Open a new Colab notebook and perform all of the steps up to *Step 3* used in *Exercise 3.02, Creating New Features from Existing Ones*.

3. The next step in the process is to get the total number of customers under each employment status. The following code groups the data based on employment status and then counts the total number of term deposits for each employment status:

```
# Getting the total counts under each job category
jobTot = bankData.groupby('job')['y']\
                 .agg(jobTot='count').reset_index()
jobTot
```

You should get the following output:

	job	jobTot
0	admin.	5171
1	blue-collar	9732
2	entrepreneur	1487
3	housemaid	1240
4	management	9458
5	retired	2264
6	self-employed	1579
7	services	4154
8	student	938
9	technician	7597
10	unemployed	1303
11	unknown	288

Figure 3.54: Employment status categories

4. Find the propensity class totals for each employment status.

   Similar to *Exercise 3.02*, *Business Hypothesis Testing for Age versus Propensity for a Term Loan* the next step is to get the total count under each class of propensity (**`yes`** and **`no`**) for each of the employment statuses. The data is grouped with respect to employment status and then with respect to the propensity, using the **`group_by()`** function. The count of the propensity is then found using the **`summarise()`** function to get the required details:

```
# Getting all the details in one place
jobProp = bankData.groupby(['job', 'y'])['y']\
                  .agg(jobCat='count').reset_index()
```

5. Get the proportionate counts for employment status.

   The next step is to get the proportionate counts, by dividing the propensity class under each employment status with the total count under each employment status. First, we merge the propensity class total DataFrame with the total count DataFrame using the **`pd.merge()`** function. Once this is done, divide the class-wise totals by the total count under the respective employment status to get the class-wise proportion:

```
# Merging both the data frames
jobComb = pd.merge(jobProp, jobTot, on=['job'])
jobComb['catProp'] = (jobComb.jobCat/jobComb.jobTot)*100
```

6. The final visualization of the proportion can be implemented using the **`matplot`** library stacked bar chart, as follows:

```
import matplotlib.pyplot as plt
import numpy as np
```

7. Create separate DataFrames for **`yes`** and **`no`**:

```
jobcombYes = jobComb[jobComb['y'] == 'yes']
jobcombNo = jobComb[jobComb['y'] == 'no']
```

8. Get the length of the *x* axis labels, as shown in the following code snippet:

```
xlabels = jobTot['job'].nunique()
```

9. Get the proportion values:

```
jobYes = jobcombYes['catProp'].unique()
jobNo = jobcombNo['catProp'].unique()
```

10. Arrange the indexes of the *x* axis (do not run the code yet):

```
ind = np.arange(xlabels)
# Get the width of each bar
width = 0.35
# Getting the plots
p1 = plt.bar(ind, jobYes, width)
p2 = plt.bar(ind, jobNo, width,bottom=jobYes)
plt.ylabel('Propensity Proportion')
plt.title('Propensity of purchase by Job')
```

11. Continuing in the same cell, define the *x* label indexes and *y* label indexes (do not run the code yet):

```
plt.xticks(ind, jobTot['job'].unique())
plt.yticks(np.arange(0, 100, 10))
```

12. In the same cell, define the **`legend`**:

```
plt.legend((p1[0], p2[0]), ('Yes', 'No'))

# To rotate the axis labels
plt.xticks(rotation=90)
plt.show()
```

Now execute this code cell. You should get the following output:

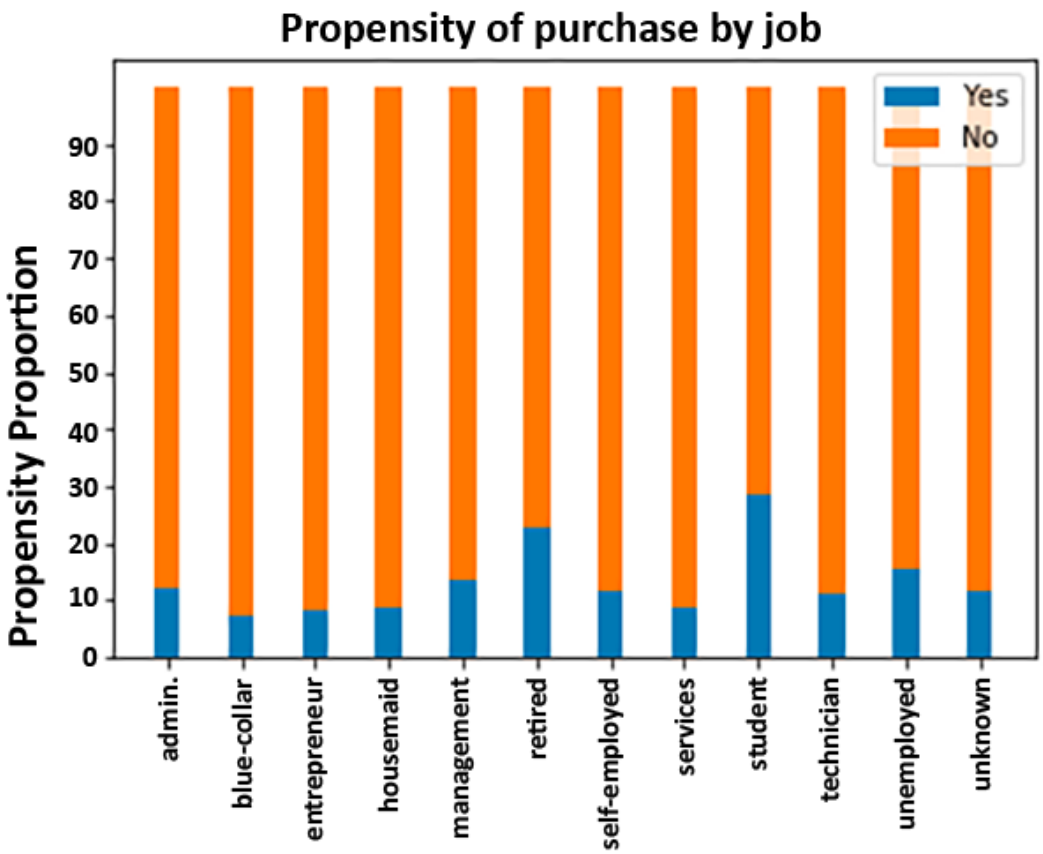

Figure 3.55: Employment status versus propensity to buy term deposits

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/34gPD3K.
>
> You can also run this example online at https://packt.live/34g8dZH.

In this activity, we have seen that the hypothesis that we formulated was wrong. In our hypothesis, we stated that individuals with higher incomes are more likely to buy term deposits. However, from the exploration, we have found out that students and retired individuals have a higher propensity for term deposits.

ACTIVITY 3.02: MODEL ITERATION 2 – LOGISTIC REGRESSION MODEL WITH FEATURE ENGINEERED VARIABLES

The following is the solution for the feature engineering activity.

Once the new features are created using some of the categorical variables such as housing and loans, we don't have to use those raw variables anymore.

SOLUTION

1. Open a new Colab notebook and perform all of the necessary steps to create the feature engineering model in *Exercise 3.04, Creating New Features from Existing Ones.*
2. Next, create the dummy variables from the remaining variables:

```
# Categorical variables, removing loan and housing
bankCat1 = pd.get_dummies(bankData\
                          [['job','marital','education',\
                            'default','contact','month',\
                            'poutcome']])
```

3. From the numerical data, we must exclude the **`balance`** variable and include the new **`assetIndex`** variable that we created:

```
bankNum1 = bankData[['age','day','duration','campaign',\
                     'pdays','previous','assetIndex']]
bankNum1.head()
```

You should get a similar output:

	age	day	duration	campaign	pdays	previous	assetIndex
0	58	5	261	1	-1	0	2.306484
1	44	5	151	1	-1	0	1.826666
2	33	5	76	1	-1	0	0.364108
3	47	5	92	1	-1	0	2.161903
4	33	5	198	1	-1	0	0.364062

Figure 3.56: Numerical variables with new feature

We can see that some of the variables such as age, day, and duration have different scales. It will be good to convert all of these variables into a common scale using the **`min max`** scaler.

4. To normalize some of the numerical variables, import the **`preprocessing`** package:

```
from sklearn import preprocessing
```

5. Create the scaling function:

```
minmaxScaler = preprocessing.MinMaxScaler()
```

6. Create the transformation variables:

```
ageT1 = bankNum1[['age']].values.astype(float)
dayT1 = bankNum1[['day']].values.astype(float)
durT1 = bankNum1[['duration']].values.astype(float)
```

7. Transform the balance data by normalizing it with **`minmaxScaler`**:

```
bankNum1['ageTran'] = minmaxScaler.fit_transform(ageT1)
bankNum1['dayTran'] = minmaxScaler.fit_transform(dayT1)
bankNum1['durTran'] = minmaxScaler.fit_transform(durT1)
```

8. Create a new numerical variable by selecting the transformed variables:

```
bankNum2 = bankNum1[['ageTran','dayTran','durTran',\
                     'campaign','pdays','previous',\
                     'assetIndex']]
```

9. Print the head of the data:

```
bankNum2.head()
```

You should get the following output:

	ageTran	dayTran	durTran	campaign	pdays	previous	assetIndex
0	0.519481	0.133333	0.053070	1	-1	0	2.306484
1	0.337662	0.133333	0.030704	1	-1	0	1.826666
2	0.194805	0.133333	0.015453	1	-1	0	0.364108
3	0.376623	0.133333	0.018707	1	-1	0	2.161903
4	0.194805	0.133333	0.040260	1	-1	0	0.364062

Figure 3.57: Transformed numerical variables

Let's now concatenate the numerical and categorical variables to create the new datasets.

10. Prepare the **X** and **Y** variables:

```
# Preparing the X variables
X = pd.concat([bankCat1, bankNum2], axis=1)
print(X.shape)
# Preparing the Y variable
Y = bankData['y']
print(Y.shape)
X.head()
```

You should get the following output:

	job_admin.	job_blue-collar	job_entrepreneur	job_housemaid	job_management	job_retired	job_self-employed	job_services	job_student	job_technician	...
0	0	0	0	0	1	0	0	0	0	0	...
1	0	0	0	0	0	0	0	0	0	1	...
2	0	0	1	0	0	0	0	0	0	0	...
3	0	1	0	0	0	0	0	0	0	0	...
4	0	0	0	0	0	0	0	0	0	0	...

5 rows × 47 columns

Figure 3.58: Concatenated features

Let's now create the training and test sets by splitting the dataset and then fitting the model on the training set.

11. Split the data into training and test sets:

```
from sklearn.model_selection import train_test_split
# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split\
                                   (X, Y, test_size=0.3, \
                                    random_state=123)
```

12. Define the **`LogisticRegression`** function:

```
from sklearn.linear_model import LogisticRegression
# Defining the LogisticRegression function
bankModel = LogisticRegression()
bankModel.fit(X_train, y_train)
```

13. Generate the predictions from the model. After generating predictions, print the accuracy of the model:

```
pred = bankModel.predict(X_test)
print('Accuracy of Logistic regression model '\
      'prediction on test set: {:.2f}'\
      .format(bankModel.score(X_test, y_test)))
```

You should get a similar output:

```
Accuracy of Logistic regression model prediction on test set: 0.89
```

Figure 3.59: Accuracy of the Logisticr regression model prediction

Here is the confusion matrix for the model:

```
from sklearn.metrics import confusion_matrix
confusionMatrix = confusion_matrix(y_test, pred)
print(confusionMatrix)
```

You should get a similar output with different values:

```
[[11738   260]
 [ 1191   375]]
```

Figure 3.60: Confusion matrix

Here are the metrics for the feature engineered model:

```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred))
```

You should get a similar output with different values:

```
              precision    recall  f1-score   support

          no       0.91      0.98      0.94     11998
         yes       0.59      0.24      0.34      1566

    accuracy                           0.89     13564
   macro avg       0.75      0.61      0.64     13564
weighted avg       0.87      0.89      0.87     13564
```

Figure 3.61: Metrics for feature engineered models

> **NOTE**
>
> The output shown may differ from the output you get on your system.

From the feature engineered variables, we can see that there is an improvement in the **`no`** cases, where the recall has improved from **`0.97`** to **`0.98`**. From the confusion matrix, we can observe that more **`no`** cases were correctly classified from what was there before. However, it has also resulted in the degradation of the **`yes`** cases. The features that were created have not given us the results that we really want. Ideally, we would have wanted to improve the metrics for the **`yes`** cases.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3lbYmu6.
>
> You can also run this example online at https://packt.live/34c1wIa.

CHAPTER 4: MULTICLASS CLASSIFICATION WITH RANDOMFOREST

ACTIVITY 4.01: TRAIN A RANDOM FOREST CLASSIFIER ON THE ISOLET DATASET

SOLUTION

1. Open a new Colab notebook.
2. Import the **`pandas`** package, **`train_test_split`**, **`RandomForestClassifier`**, and **`accuracy_score`** from **`sklearn`**:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

3. Create a variable called **`file_url`** that contains the URL to the dataset:

```
file_url = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter04/Dataset/phpB0xrNj.csv'
```

4. Load the dataset into a DataFrame using the **`.read_csv()`** method from **`pandas`**:

```
df = pd.read_csv(file_url)
```

5. Print the first five rows using the **`.head()`** method:

```
df.head()
```

You should get the following output:

	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12
0	-0.4394	-0.0930	0.1718	0.4620	0.6226	0.4704	0.3578	0.0478	-0.1184	-0.2310	-0.2958	-0.2704
1	-0.4348	-0.1198	0.2474	0.4036	0.5026	0.6328	0.4948	0.0338	-0.0520	-0.1302	-0.0964	-0.2084
2	-0.2330	0.2124	0.5014	0.5222	-0.3422	-0.5840	-0.7168	-0.6342	-0.8614	-0.8318	-0.7228	-0.6312
3	-0.3808	-0.0096	0.2602	0.2554	-0.4290	-0.6746	-0.6868	-0.6650	-0.8410	-0.9614	-0.7374	-0.7084
4	-0.3412	0.0946	0.6082	0.6216	-0.1622	-0.3784	-0.4324	-0.4358	-0.4966	-0.5406	-0.5472	-0.5440

5 rows × 618 columns

Figure 4.44: First five rows of df

6. Extract the **`class`** target variable into a new variable called **`y`** using the **`.pop()`** method:

```
y = df.pop('class')
```

7. Split the data into training and testing sets with **`train_test_split()`** and the **`test_size=0.3`** and **`random_state=888`** parameters:

```
X_train, X_test, y_train, y_test = train_test_split\
                                   (df, y, test_size=0.3, \
                                    random_state=888)
```

8. Create a function called **`train_rf`** that will take the following arguments, instantiate a **`RandomForestClassifier`** with these arguments, and fit it with the training set: **`X_train`**, **`y_train`**, **`random_state=888`**, **`n_estimators=10`**, **`max_depth=None`**, **`min_samples_leaf=1`**, **`max_features='sqrt'`**

```
def train_rf(X_train, y_train, random_state=888, \
             n_estimators=10, max_depth=None, \
             min_samples_leaf=1, max_features='sqrt'):
    rf_model = RandomForestClassifier\
               (random_state=random_state, \
                n_estimators=n_estimators, \
                max_depth=max_depth, \
                min_samples_leaf=min_samples_leaf, \
                max_features=max_features)
    rf_model.fit(X_train, y_train)
    return rf_model
```

9. Call the **`train_rf`** function with the training set and save it in a variable called **`rf_1`**, then print the model hyperparameters with the **`.get_params()`** method:

```
rf_1 = train_rf(X_train, y_train)
rf_1.get_params()
```

You should get the following output:

```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'sqrt',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 10,
 'n_jobs': None,
 'oob_score': False,
 'random_state': 888,
 'verbose': 0,
 'warm_start': False}
```

Figure 4.45: Hyperparameters of rf_1

10. Create a function called **`get_preds`** that will take as arguments **`rf_model`**, **`X_train`**, and **`X_test`**. This function will predict using the **`.predict()`** method on the training and testing sets and return the results:

```
def get_preds(rf_model, X_train, X_test):
    train_preds = rf_model.predict(X_train)
    test_preds = rf_model.predict(X_test)
    return train_preds, test_preds
```

11. Call the **get_preds** function with **rf_1**, **X_train**, and **X_test** set and save the results into two variables called **trn_preds** and **tst_preds**:

```
trn_preds, tst_preds = get_preds(rf_1, X_train, X_test)
```

12. Create a function called **print_accuracy** that will take as arguments **y_train**, **y_test**, **train_preds**, and **test_preds.** This function will calculate the accuracy scores for the training and testing sets, and print and return those scores:

```
def print_accuracy(y_train, y_test, train_preds, \
                   test_preds):
    train_acc = accuracy_score(y_train, train_preds)
    test_acc = accuracy_score(y_test, test_preds)
    print(train_acc)
    print(test_acc)
    return train_acc, test_acc
```

13. Call the **print_accuracy** function with the target variable of the training and testing sets and the predictions, and save the results in two variables called **trn_acc** and **tst_acc**:

```
trn_acc, tst_preds = print_accuracy(y_train, y_test, \
                                    trn_preds, tst_preds)
```

You should get the following output:

```
0.9981674912955837
0.8935897435897436
```

Figure 4.46: Accuracy scores of the training and testing sets

The difference between the training and testing sets' accuracy scores is quite high so our model is overfitting.

14. Create a function called **fit_predict_rf** that will take the following arguments: **X_train**, **y_train**, **random_state=888**, **n_estimators=10**, **max_depth=None**, **min_samples_leaf=1**, and **max_features='sqrt'**. This function will call the three functions you created before (**train_rf**, **get_preds**, and **print_accuracy**) and return the model trained, predictions, and accuracy scores for the training and testing sets:

```
def fit_predict_rf(X_train, X_test, y_train, y_test, \
                   random_state=888, n_estimators=10, \
```

```
                    max_depth=None, min_samples_leaf=1, \
                    max_features='sqrt'):
    rf_model = train_rf(X_train, y_train, \
                        random_state=random_state, \
                        n_estimators=n_estimators, \
                        max_depth=max_depth, \
                        min_samples_leaf=min_samples_leaf, \
                        max_features=max_features)
    train_preds, test_preds = get_preds(rf_model, X_train, \
                                        X_test)
    train_acc, test_acc = print_accuracy(y_train, y_test, \
                                         train_preds, \
                                         test_preds)
    return rf_model, train_preds, test_preds, \
           train_acc, test_acc
```

15. Call the **`fit_predict`** function with the following arguments: **`X_train`**, **`X_test`**, **`y_train`**, **`y_test`**, **`random_state=888`**, **`n_estimators=20`**, **`max_depth=None`**, **`min_samples_leaf=1`**, and **`max_features='sqrt'`**. Save the results in variables called **`rf_model_1`**, **`trn_preds_1`**, **`tst_preds_1`**, **`trn_acc_1`**, and **`tst_acc_1`**:

```
rf_model_1, trn_preds_1, tst_preds_1, \
trn_acc_1, tst_acc_1 = fit_predict_rf(X_train, X_test, y_train,\
                                      y_test, random_state=888,\
                                      n_estimators=20, \
                                      max_depth=None, \
                                      min_samples_leaf=1, \
                                      max_features='sqrt')
```

You should get the following output:

```
0.9998167491295583
0.9192307692307692
```

Figure 4.47: Accuracy scores of the training and testing sets for rf_model_1

Compared to the previous results, our model is overfitting less. We slightly increased the accuracy score of the testing set and therefore decreased the difference with the training set.

16. Call the **fit_predict** function with the following arguments: **X_train**, **X_test**, **y_train**, **y_test**, **random_state=888**, **n_estimators=50**, **max_depth=None**, **min_samples_leaf=1**, and **max_features='sqrt'**. Save the results in variables called **rf_model_1**, **trn_preds_1**, **tst_preds_1**, **trn_acc_1**, and **tst_acc_1**:

```
rf_model_2, trn_preds_2, tst_preds_2, \
trn_acc_2, tst_acc_2 = fit_predict_rf(X_train, X_test, y_train,\
                                      y_test, random_state=888,\
                                      n_estimators=50, \
                                      max_depth=None, \
                                      min_samples_leaf=1, \
                                      max_features='sqrt')
```

You should get the following output:

```
1.0
0.9333333333333333
```

Figure 4.48: Accuracy scores of the training and testing sets for rf_model_2

With this set of hyperparameters, we obtained a better accuracy score for the training and testing sets but our model is still overfitting.

17. Call the **fit_predict** function with the following arguments: **X_train**, **X_test**, **y_train**, **y_test**, **random_state=888**, **n_estimators=50**, **max_depth=5**, **min_samples_leaf=1**, and **max_features='sqrt'**. Save the results in variables called **rf_model_3**, **trn_preds_3**, **tst_preds_3**, **trn_acc_3**, and **tst_acc_3**:

```
rf_model_3, trn_preds_3, tst_preds_3, \
trn_acc_3, tst_acc_3 = fit_predict_rf(X_train, X_test, y_train,\
                                      y_test, random_state=888,\
                                      n_estimators=50, \
                                      max_depth=5, \
                                      min_samples_leaf=1, \
                                      max_features='sqrt')
```

You should get the following output:

```
0.8552318123511087
0.8213675213675213
```

Figure 4.49: Accuracy scores of the training and testing sets for rf_model_3

Now the model performance has decreased drastically but the model is not overfitting much.

18. Call the **`fit_predict`** function with the following arguments: **`X_train`**, **`X_test`**, **`y_train`**, **`y_test`**, **`random_state=888`**, **`n_estimators=50`**, **`max_depth=10`**, **`min_samples_leaf=1`**, and **`max_features='sqrt'`**. Save the results in variables called **`rf_model_4`**, **`trn_preds_4`**, **`tst_preds_4`**, **`trn_acc_4`**, and **`tst_acc_4`**:

```
rf_model_4, trn_preds_4, tst_preds_4, \
trn_acc_4, tst_acc_4 = fit_predict_rf(X_train, X_test, y_train,\
                                      y_test, random_state=888,\
                                      n_estimators=50, \
                                      max_depth=10, \
                                      min_samples_leaf=1, \
                                      max_features='sqrt')
```

You should get the following output:

```
0.9844236760124611
0.9260683760683761
```

Figure 4.50: Accuracy scores of the training and testing sets for rf_model_4

The accuracy scores for the training and testing sets are quite high but their difference is still significant.

19. Call the **`fit_predict`** function with the following arguments: **`X_train`**, **`X_test`**, **`y_train`**, **`y_test`**, **`random_state=888`**, **`n_estimators=50`**, **`max_depth=10`**, **`min_samples_leaf=10`**, and **`max_features='sqrt'`**. Save the results in variables called **`rf_model_5`**, **`trn_preds_5`**, **`tst_preds_5`**, **`trn_acc_5`**, and **`tst_acc_5`**:

```
rf_model_5, trn_preds_5, tst_preds_5, \
trn_acc_5, tst_acc_5 = fit_predict_rf(X_train, X_test, y_train,\
                                      y_test, random_state=888,\
```

```
                                        n_estimators=50, \
                                        max_depth=10, \
                                        min_samples_leaf=10, \
                                        max_features='sqrt')
```

You should get the following output:

```
0.9622503206890233
0.9192307692307692
```

Figure 4.51: Accuracy score of the training and testing sets for rf_model_5

With this set of hyperparameters, we are starting to reach a good compromise between performance and overfitting.

20. Call the **`fit_predict`** function with the following arguments: **`X_train`**, **`X_test`**, **`y_train`**, **`y_test`**, **`random_state=888`**, **`n_estimators=50`**, **`max_depth=10`**, **`min_samples_leaf=50`**, and **`max_features='sqrt'`**. Save the results in variables called **`rf_model_6`**, **`trn_preds_6`**, **`tst_preds_6`**, **`trn_acc_6`**, and **`tst_acc_6`**:

```
rf_model_6, trn_preds_6, tst_preds_6, \
trn_acc_6, tst_acc_6 = fit_predict_rf(X_train, X_test, y_train,\
                                        y_test, random_state=888,\
                                        n_estimators=50, \
                                        max_depth=10, \
                                        min_samples_leaf=50, \
                                        max_features='sqrt')
```

You should get the following output:

```
0.9184533626534725
0.8940170940170941
```

Figure 4.52: Accuracy scores of the training and testing sets for rf_model_6

Now the accuracy scores for the training and testing sets are quite close to each other and we still have a good level of performance (around **`0.9`**).

21. Call the **fit_predict** function with the following arguments: **X_train**, **X_test**, **y_train**, **y_test**, **random_state=888**, **n_estimators=50**, **max_depth=10**, **min_samples_leaf=50**, and **max_features=0.5**. Save the results in variables called **rf_model_7**, **trn_preds_7**, **tst_preds_7**, **trn_acc_7**, and **tst_acc_7**:

```
rf_model_7, trn_preds_7, tst_preds_7, \
trn_acc_7, tst_acc_7 = fit_predict_rf(X_train, X_test, y_train,\
                                      y_test, random_state=888,\
                                      n_estimators=50, \
                                      max_depth=10, \
                                      min_samples_leaf=50, \
                                      max_features=0.5)
```

You should get the following output:

```
0.8926149899212021
0.867948717948718
```

Figure 4.53: Accuracy scores of the training and testing sets for rf_model_7

Here, our model is not achieving the same level as for the previous result.

22. Call the **fit_predict** function with the following arguments: **X_train**, **X_test**, **y_train**, **y_test**, **random_state=888**, **n_estimators=50**, **max_depth=10**, **min_samples_leaf=50**, and **max_features=0.3**. Save the results in variables called **rf_model_8**, **trn_preds_8**, **tst_preds_8**, **trn_acc_8**, and **tst_acc_8**:

```
rf_model_8, trn_preds_8, tst_preds_8, \
trn_acc_8, tst_acc_8 = fit_predict_rf(X_train, X_test, y_train,\
                                      y_test, random_state=888,\
                                      n_estimators=50, \
                                      max_depth=10, \
                                      min_samples_leaf=50, \
                                      max_features=0.3)
```

You should get the following output:

```
0.9008612790910757
0.8717948717948718
```

Figure 4.54: Accuracy scores of the training and testing sets for rf_model_8

This final set of hyperparameters still doesn't achieve better results than the one we find with **`n_estimators=50`**, **`max_depth=10`**, **`min_samples_leaf=50`**, **`max_features=0.5`**.

We built several RandomForest classifier models that accurately predict the letters spoken from audio signals. We tried several values for the hyperparameters **`n_estimators`**, **`max_depth`**, **`min_samples_leaf`**, and **`max_features`**. The best combination of hyperparameters we came up with is **`n_estimators=50`**, **`max_depth=10`**, **`min_samples_leaf=50`**, and **`max_features='sqrt'`**.

We achieved a final accuracy score of **`0.92`** for the training set and **`0.89`** for the testing set. The model is still overfitting slightly and could still be improved but it is a remarkable result given that it was our first attempt.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3aFdk6M.
>
> You can also run this example online at https://packt.live/2Yj66k3.

CHAPTER 5: PERFORMING YOUR FIRST CLUSTER ANALYSIS

ACTIVITY 5.01: PERFORM CUSTOMER SEGMENTATION ANALYSIS IN A BANK USING K-MEANS

SOLUTION

1. Open a new Colab notebook.
2. Now **`import`** the important libraries of **`pandas`**, **`Kmeans`**, **`altair`**, and **`sklearn`** (**`KMeans`** and **`StandardScaler`**):

```
import pandas as pd
from sklearn.cluster import KMeans
import altair as alt
from sklearn.preprocessing import StandardScaler
```

3. Assign the link to the dataset to a variable called **`file_url`**:

```
file_url = 'https://raw.githubusercontent.com'\
           '/TrainingByPackt/The-Data-Science-Workshop'\
           '/master/Chapter05/DataSet'\
           '/german.data-numeric'
```

4. Load the dataset using the **`read_csv()`** method from the **`pandas`** package and the following parameters: **`header=None`**, **`sep= '\s\s+'`** and **`prefix='X'`**:

```
df = pd.read_csv(file_url, header=None, \
                 sep='\s\s+', prefix='X')
```

5. Display the first five rows of the DataFrame:

```
df.head()
```

You should get the following output:

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15	X16	X17	X18	X19	X20	X21	X22	X23	X24
0	1	6	4	12	5	5	3	4	1	67	3	2	1	2	1	0	0	1	0	0	1	0	0	1	1.0
1	2	48	2	60	1	3	2	2	1	22	3	1	1	1	1	0	0	1	0	0	1	0	0	1	2.0
2	4	12	4	21	1	4	3	3	1	49	3	1	2	1	1	0	0	1	0	0	1	0	1	0	1.0
3	1	42	2	79	1	4	3	4	2	45	3	1	2	1	1	0	0	0	0	0	0	0	0	1	1.0
4	1	24	3	49	1	3	3	4	4	53	3	2	2	1	1	1	0	1	0	0	0	0	0	1	2.0

Figure 5.56: The first five rows of the dataset

6. Extract the **X3** and **X9** columns and assign them to a new variable called **X**:

```
X = df[['X3', 'X9']]
```

7. Instantiate a **StandardScaler** object, standardize the data, and store the result in a variable called **X_scaled**:

```
standard_scaler = StandardScaler()
X_scaled = standard_scaler.fit_transform(X)
```

8. Create an empty pandas DataFrame called **clusters** and an empty list called **inertia**:

```
clusters = pd.DataFrame()
inertia = []
```

9. Create a new column called **'cluster_range'** in the **clusters** DataFrame and assign a range from **1** to **15**:

```
clusters['cluster_range'] = range(1, 15)
```

10. Using a **for** loop, fit a k-means model with the number of clusters defined in the **'cluster_range'** column, extract the relevant **inertia** value, and append it to the **inertia** list:

```
for k in clusters['cluster_range']:
    kmeans = KMeans(n_clusters=k, random_state=8).fit(X_scaled)
    inertia.append(kmeans.inertia_)
```

11. Create a new column called **'cluster_range'** from the **clusters** DataFrame and assign it the **inertia** list:

```
clusters['inertia'] = inertia
```

12. Print the **`clusters`** DataFrame:

```
clusters
```

You should get the following output:

	cluster_range	inertia
0	1	2000.000000
1	2	1280.612749
2	3	767.637196
3	4	576.086134
4	5	443.905649
5	6	360.418261
6	7	291.393050
7	8	252.709449
8	9	219.498996
9	10	193.015983
10	11	174.554284
11	12	159.035228
12	13	147.613182
13	14	135.283361

Figure 5.57: Inertia values for each number of clusters

13. Use the **`altair`** package and the **`mark_line`** and **`encode`** methods to display the Elbow plot:

```
alt.Chart(clusters).mark_line()\
                    .encode(alt.X('cluster_range'), \
                            alt.Y('inertia'))
```

You should get the following output:

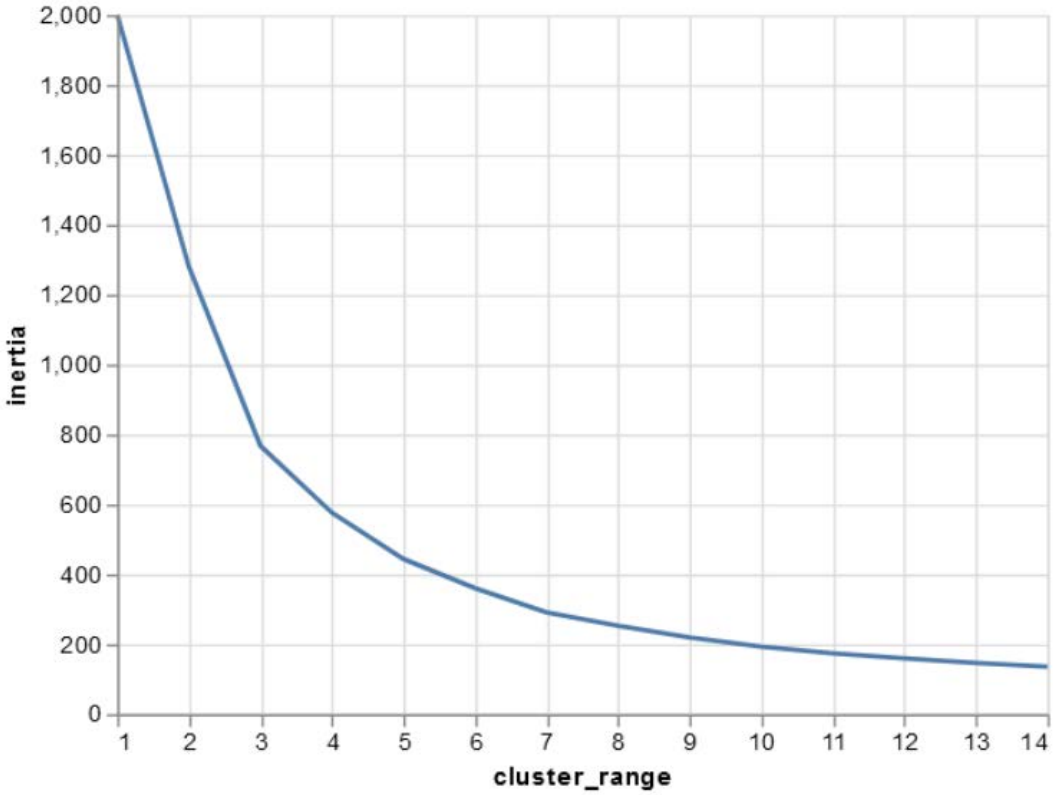

Figure 5.58: The Elbow plot and the optimal number of clusters

14. Looking at the Elbow plot, find the optimal number of clusters and save this value in a new variable called **`clusters_number`**:

```
clusters_number = 5
```

15. Fit a k-means++ algorithm with this number of clusters, **`n_init=50`**, and **`max_iter=1000`**:

```
kmeans = KMeans(random_state=1, n_clusters=clusters_number, \
                init='k-means++', n_init=50, max_iter=1000)
kmeans.fit(X_scaled)
```

16. Use the **`predict()`** method from **`sklearn`** to get the assigned clusters for all data points saved in **`X_scaled`**:

```
df['cluster'] = kmeans.predict(X_scaled)
```

17. Plot the scatter plot with the **`altair`** package:

```
scatter_plot = alt.Chart(df).mark_circle()
scatter_plot.encode(x='X3', y='X9',color='cluster:N')
```

You should get the following output:

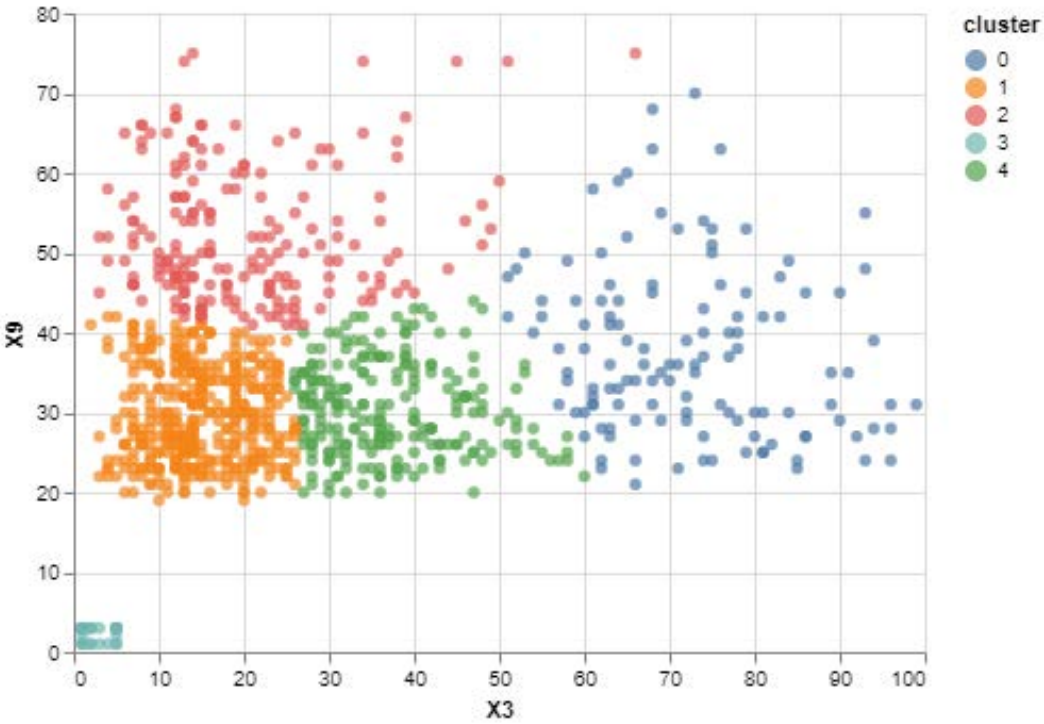

Figure 5.59: Scatter plot of the four clusters found

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3geDrTh.
>
> This section does not currently have an online interactive example. You can try this code on Google Colab

CHAPTER 6: HOW TO ASSESS PERFORMANCE

ACTIVITY 6.01: TRAIN THREE DIFFERENT MODELS AND USE EVALUATION METRICS TO PICK THE BEST PERFORMING MODEL

SOLUTION

1. Open a Colab notebook.
2. Load the necessary libraries:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
```

3. Read in the data:

```
df = pd.read_csv('https://raw.githubusercontent.com'\
                 '/PacktWorkshops/The-Data-Science-Workshop'\
                 '/master/Chapter06/Dataset'\
                 '/bank-additional-full.csv', sep=';')
```

4. Explore the data:

```
df.info()
```

The output should be similar to the following:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
age               41188 non-null int64
job               41188 non-null object
marital           41188 non-null object
education         41188 non-null object
default           41188 non-null object
housing           41188 non-null object
loan              41188 non-null object
contact           41188 non-null object
month             41188 non-null object
day_of_week       41188 non-null object
duration          41188 non-null int64
campaign          41188 non-null int64
pdays             41188 non-null int64
previous          41188 non-null int64
poutcome          41188 non-null object
emp.var.rate      41188 non-null float64
cons.price.idx    41188 non-null float64
cons.conf.idx     41188 non-null float64
euribor3m         41188 non-null float64
nr.employed       41188 non-null float64
y                 41188 non-null object
dtypes: float64(5), int64(5), object(11)
memory usage: 6.6+ MB
```

Figure 6.47: Inspecting the dataset

You can run the following code to see the top five rows of the dataframe:

```
df.head()
```

The output should be similar to the following:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	campaign	pdays	previous	poutcome	emp.var.rate
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.1
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.1
2	37	services	married	high.school	no	yes	no	telephone	may	mon	...	1	999	0	nonexistent	1.1
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	...	1	999	0	nonexistent	1.1
4	56	services	married	high.school	no	no	yes	telephone	may	mon	...	1	999	0	nonexistent	1.1

5 rows × 21 columns

Figure 6.48: The top five rows of the dataframe

5. Convert categorical variables using **`pd.get_dummies()`**:

```
cat_cols = ['job', 'marital', 'education', 'default', \
            'housing', 'loan', 'contact', 'month', \
            'day_of_week', 'poutcome']
_df = pd.get_dummies(df, columns=cat_cols, \
                     prefix=cat_cols, drop_first=True)
_df.info()
```

A truncated output is as follows:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 54 columns):
age                             41188 non-null int64
duration                        41188 non-null int64
campaign                        41188 non-null int64
pdays                           41188 non-null int64
previous                        41188 non-null int64
emp.var.rate                    41188 non-null float64
cons.price.idx                  41188 non-null float64
cons.conf.idx                   41188 non-null float64
euribor3m                       41188 non-null float64
nr.employed                     41188 non-null float64
y                               41188 non-null object
job_blue-collar                 41188 non-null uint8
job_entrepreneur                41188 non-null uint8
job_housemaid                   41188 non-null uint8
job_management                  41188 non-null uint8
job_retired                     41188 non-null uint8
job_self-employed               41188 non-null uint8
job_services                    41188 non-null uint8
job_student                     41188 non-null uint8
job_technician                  41188 non-null uint8
job_unemployed                  41188 non-null uint8
```

Figure 6.49: Converting the categorical variables

You can run the following code to see the top five rows of the dataframe:

```
_df.head()
```

The output should be similar to the following:

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	...	month_may	month_nov
0	56	261	1	999	0	1.1	93.994	-36.4	4.857	5191.0	...	1	0
1	57	149	1	999	0	1.1	93.994	-36.4	4.857	5191.0	...	1	0
2	37	226	1	999	0	1.1	93.994	-36.4	4.857	5191.0	...	1	0
3	40	151	1	999	0	1.1	93.994	-36.4	4.857	5191.0	...	1	0
4	56	307	1	999	0	1.1	93.994	-36.4	4.857	5191.0	...	1	0

5 rows × 54 columns

Figure 6.50: The top five rows of the dataframe

6. Prepare the **`X`** and **`y`** variables:

```
X = _df.drop(['y'], axis=1)
X = X.values
y = df['y'].apply(lambda x: 0 if x == 'no' else 1)
y = y.values
```

7. Split the data into training and evaluation sets:

```
train_X, eval_X, train_y, eval_y = train_test_split\
                                   (X, y, test_size=0.3, \
                                    random_state=0)
val_X, test_X, val_y, test_y = train_test_split\
                                (eval_X, eval_y, random_state=0)
```

8. Create an instance of **`LogisticRegression`**:

```
lr_model = LogisticRegression()
```

9. Fit the training data to the logistic regression model:

```
lr_model.fit(train_X, train_y)
```

10. Use the evaluation set to make a prediction:

```
lr_preds = lr_model.predict(val_X)
```

11. Use the prediction from the logistic regression model to compute the classification report:

```
lr_report = classification_report(val_y, lr_preds)
print(lr_report)
```

The output should be similar to the following:

```
              precision    recall  f1-score   support

           0       0.93      0.97      0.95      8220
           1       0.65      0.41      0.50      1047

    accuracy                           0.91      9267
   macro avg       0.79      0.69      0.73      9267
weighted avg       0.90      0.91      0.90      9267
```

Figure 6.51: Classification report

12. Create an instance of **`DecisionTreeClassifier`**:

```
dt_model = DecisionTreeClassifier(max_depth= 6)
```

13. Fit the training data to the decision tree classifier model:

```
dt_model.fit(train_X, train_y)
```

14. Using the decision tree classifier model, make a prediction on the evaluation dataset:

```
dt_preds = dt_model.predict(val_X)
```

15. Use the prediction from the decision tree classifier model to compute the classification report:

```
dt_report = classification_report(val_y, dt_preds)
print(dt_report)
```

The output should be similar to the following:

```
              precision    recall  f1-score   support

           0       0.94      0.96      0.95      8220
           1       0.66      0.54      0.60      1047

    accuracy                           0.92      9267
   macro avg       0.80      0.75      0.78      9267
weighted avg       0.91      0.92      0.91      9267
```

Figure 6.52: Classification report

16. Compare the classification report from the linear regression model and the classification report from the decision tree classifier model to determine which is the better model.

17. Create an instance of **`RandomForestClassifier`**:

```
rf_model = RandomForestClassifier(n_estimators=1000)
```

18. Fit the training data to the random forest classifier model:

```
rf_model.fit(train_X, train_y)
```

19. Using the random forest classifier model, make a prediction on the evaluation dataset:

```
rf_preds = rf_model.predict(val_X)
```

20. Using the prediction from the random forest classifier, compute the classification report:

```
rf_report = classification_report(val_y, rf_preds)
print(rf_report)
```

The output should be similar to the following:

```
              precision    recall  f1-score   support

           0       0.94      0.97      0.95      8220
           1       0.68      0.48      0.56      1047

    accuracy                           0.92      9267
   macro avg       0.81      0.72      0.76      9267
weighted avg       0.91      0.92      0.91      9267
```

Figure 6.53: Classification report

21. Compare the classification report from the linear regression model to the classification report from the random forest classifier to decide which model to keep or improve upon.

    Compare the R2 scores of all three models:

```
print('Linear Score: {}, DecisionTree Score: {}, '\
      'RandomForest Score: {}'\
      .format(lr_model.score(val_X, val_y), \
              dt_model.score(val_X, val_y), \
              rf_model.score(val_X, val_y)))
```

    The output should be similar to the following:

```
Linear Score: 0.9087083198446099, DecisionTree Score: 0.917125283263192, RandomForest Score: 0.9153987266645
```

Figure 6.54: Comparing the R2 scores

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2YhquSI.
>
> You can also run this example online at https://packt.live/3g84orS.

You can now compare the R2 scores of the three models that you have trained and see from the result that the decision tree produced a marginally better score of 0.917 for the particular training data that you used.

CHAPTER 7: THE GENERALIZATION OF MACHINE LEARNING MODELS

ACTIVITY 7.01: FIND AN OPTIMAL MODEL FOR PREDICTING THE CRITICAL TEMPERATURES OF SUPERCONDUCTORS

SOLUTION

1. Open a Colab notebook.
2. Load the necessary libraries:

```
import pandas as pd
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler, \
PolynomialFeatures
```

3. Read in the data from the **`superconduct`** folder:

```
_df = pd.read_csv\
      ('https://raw.githubusercontent.com/PacktWorkshops'\
       '/The-Data-Science-Workshop/master/Chapter07/Dataset'\
       '/superconduct/train.csv')
_df.info()
```

The output will be as follows:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21263 entries, 0 to 21262
Data columns (total 82 columns):
number_of_elements                  21263 non-null int64
mean_atomic_mass                    21263 non-null float64
wtd_mean_atomic_mass                21263 non-null float64
gmean_atomic_mass                   21263 non-null float64
wtd_gmean_atomic_mass               21263 non-null float64
entropy_atomic_mass                 21263 non-null float64
wtd_entropy_atomic_mass             21263 non-null float64
range_atomic_mass                   21263 non-null float64
wtd_range_atomic_mass               21263 non-null float64
std_atomic_mass                     21263 non-null float64
wtd_std_atomic_mass                 21263 non-null float64
mean_fie                            21263 non-null float64
wtd_mean_fie                        21263 non-null float64
gmean_fie                           21263 non-null float64
wtd_gmean_fie                       21263 non-null float64
entropy_fie                         21263 non-null float64
wtd_entropy_fie                     21263 non-null float64
range_fie                           21263 non-null float64
wtd_range_fie                       21263 non-null float64
std_fie                             21263 non-null float64
wtd_std_fie                         21263 non-null float64
mean_atomic_radius                  21263 non-null float64
wtd_mean_atomic_radius              21263 non-null float64
gmean_atomic_radius                 21263 non-null float64
```

Figure 7.81: Information on the dataframe

> **NOTE**
>
> The output is truncated for presentation purposes. The complete output is available on the GitHub repository here: https://packt.live/30l7Gfg

4. Prepare the **X** and **y** variables:

```
X = _df.drop(['critical_temp'], axis=1).values
y = _df['critical_temp'].values
```

5. Split the data into training and evaluation sets:

```
train_X, eval_X, train_y, eval_y = train_test_split\
                                   (X, y, test_size=0.8, \
                                    random_state=0)
```

6. Create a baseline linear regression model:

```
model_1 = LinearRegression()
```

7. Fit the model to the training data:

```
model_1.fit(train_X, train_y)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Figure 7.82: Fitting the model

8. Print out the **R2** score and **MSE** of the model:

```
print('Model 1 R2 Score: {}'\
      .format(model_1.score(eval_X, eval_y)))
```

The output will be as follows:

```
Model 1 R2 Score: 0.7328447712730708
```

Figure 7.83: The R2 score and MSE of the model

Similarly, print the model coefficients.

```
print(model_1.coef_)
```

The output will be as follows:

```
[-4.94346491e+00  8.70902756e-01 -9.98866501e-01 -5.83760774e-01
  7.93953012e-01 -2.56095021e+01 -5.50632648e+00  1.27121023e-01
 -3.64947050e-02 -2.33513663e-01 -3.17000184e-02  3.10498264e-01
 -2.70341636e-01 -3.27264900e-01  3.01282843e-01 -8.31664513e+01
  4.76913648e+01  8.40711399e-02  2.07050106e-02 -2.97568138e-01
  3.97698378e-02 -1.06142981e+00  3.61994102e+00  6.70321650e-01
 -3.22006412e+00  4.02729358e+01  4.97586647e+01  1.91181858e-01
 -8.63617280e-02 -1.01111516e-01 -6.03489660e-01 -5.08255800e-03
  1.01228657e-03  2.34243690e-03  6.57781429e-04  1.24587526e+01
 -1.10269822e+01 -1.30154742e-03  5.73536446e-04  4.95127593e-03
 -9.30322207e-04  5.82173815e-02  4.77320536e-01  5.07267163e-02
 -5.53776412e-01  5.02250052e+00 -1.96605153e+01 -3.58831794e-01
 -1.02331087e-01  1.14617678e+00 -5.43201942e-01  1.66517118e+00
 -1.86511975e+00 -1.18487297e+00  1.27906539e+00 -2.20915415e+01
  2.34252719e+01 -4.11107209e-01  7.35352108e-01 -5.28505687e-01
  7.74281630e-01 -1.78504148e-01  5.87315418e-01  4.71295603e-02
 -4.18163312e-01  7.81304600e+00  6.87307727e+00 -2.97522968e-02
 -2.06281413e-01  2.10538634e-01 -3.98675109e-02 -5.52756012e+00
  8.24378449e+00  7.98555157e+00 -1.16791127e+01  8.10069280e+01
 -8.54657430e+01  4.87898207e+00 -2.42910779e+00  3.46557465e+00
 -2.07905810e+01]
```

Figure 7.84: Model coefficients

Similarly, print the **MSE** for the model:

```
preds_1 = model_1.predict(eval_X)
print('Model 1 MSE: {}'\
      .format(mean_squared_error(eval_y, preds_1)))
```

The output will be as follows:

```
Model 1 MSE: 314.1265890122019
```

Figure 7.85: The MSE of model_1

9. Create a pipeline to engineer polynomial features and train a linear regression model:

```
steps = [('scaler', MinMaxScaler()),\
         ('poly', PolynomialFeatures(degree=3, \
                                     interaction_only=True)),\
         ('model', LinearRegression())]
model_2 = Pipeline(steps)
model_2.fit(train_X, train_y)
```

This step may take some time to complete. The output will be as follows:

```
Pipeline(memory=None,
         steps=[('scaler', MinMaxScaler(copy=True, feature_range=(0, 1))),
                ('poly',
                 PolynomialFeatures(degree=3, include_bias=True,
                                    interaction_only=True, order='C')),
                ('model',
                 LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                                  normalize=False))],
         verbose=False)
```

Figure 7.86: Training a linear regression model

10. Print out the R2 score and MSE:

```
print('Model 2 R2 Score: {}'\
      .format(model_2.score(eval_X, eval_y)))
```

The output will be similar to the following:

```
Model 2 R2 Score: -3.412098915405949e+18
```

Figure 7.87: The R2 score and MSE of model_2

11. Determine that this new model is overfitting. The first model had a score of **`0.73`**. You need a model with a higher score. The second model has a score of -**`3.412e+18`**, which is significantly worse. The second model is overfitting.

12. Create a pipeline to engineer polynomial features and train a ridge or lasso model:

```
steps = [('scaler', MinMaxScaler()),\
         ('poly', PolynomialFeatures(degree=3, \
                                     interaction_only=True)),\
         ('model', Lasso(alpha=0.001, max_iter=2000))]
```

```
lasso_model = Pipeline(steps)
lasso_model.fit(train_X, train_y)
```

13. Print out the R2 score and MSE:

```
print('Lasso Model R2 Score: {}'\
      .format(lasso_model.score(eval_X, eval_y)))
```

The output will be as follows:

```
Lasso Model R2 Score: 0.8325230040978594
```

Figure 7.88: The R2 score and MSE of the ridge model

Print the model coefficients

```
print(lasso_model[-1].coef_[:30])
```

The output will be as follows:

```
[ 0.00000000e+00  8.74340500e-02 -7.95095837e+00 -1.30139088e-01
 -0.00000000e+00  0.00000000e+00  0.00000000e+00  3.38565726e+01
  0.00000000e+00 -0.00000000e+00 -4.13763260e+00 -2.65279487e-02
 -0.00000000e+00 -0.00000000e+00 -0.00000000e+00 -0.00000000e+00
 -0.00000000e+00  1.22329305e+01  0.00000000e+00 -0.00000000e+00
 -0.00000000e+00 -1.12633645e+01  0.00000000e+00  0.00000000e+00
  0.00000000e+00  0.00000000e+00 -0.00000000e+00  0.00000000e+00
  0.00000000e+00 -9.08364155e+00]
```

Figure 7.89: The coefficients for the ridge model

14. Determine that this model is no longer overfitting. The score is now back up to **`0.8325`**. You can also see from the first few coefficients that the magnitudes are now below a magnitude of 100. This is a good model; it's the model you could put into production.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/30l7Gfg.
>
> This section does not currently have an online interactive example. You can try this code on Google Colab

CHAPTER 8: HYPERPARAMETER TUNING

ACTIVITY 8.01: IS THE MUSHROOM POISONOUS?

SOLUTION

1. Load the data into Python using the **`pandas.read_csv()`** method, calling the object **`mushrooms`**.

```
import pandas as pd
import numpy as np
mushrooms = pd.read_csv\
            ('https://raw.githubusercontent.com'\
             '/PacktWorkshops/The-Data-Science-Workshop'\
             '/master/Chapter08/Dataset'\
             '/agaricus-lepiota.data', header=None)
```

2. Separate the target **`y`** and features **`X`** from the dataset:

```
y_raw = mushrooms.iloc[:,0]
X_raw = mushrooms.iloc[:,1:]
```

3. Recode the target **`y`** so that poisonous mushrooms are represented as **`1`** and edible mushrooms as **`0`**:

```
y = (y_raw == 'p') * 1
```

The featureset **`X`** will need to have its columns transformed into a **`numpy`** array with a binary representation. This is known as one-hot encoding:

```
from sklearn import preprocessing
encoder = preprocessing.OneHotEncoder()
encoder.fit(X_raw)
X = encoder.transform(X_raw).toarray()
```

4. Conduct both a grid and random search to find an optimal hyperparameterization for a random forest classifier. Use accuracy as your method of model evaluation. Make sure that when you initialize the classifier and when you conduct your random search, **`random_state = 100`**:

```
#Initialize the classifier.
from sklearn import ensemble
rfc = ensemble.RandomForestClassifier(n_estimators=100, \
                                      random_state=100)
```

```
#Conduct a grid search.
from sklearn import model_selection
grid = {'criterion': ['gini', 'entropy'],\
        'max_features': [2, 4, 6, 8, 10, 12, 14]}
gscv = model_selection.GridSearchCV(estimator=rfc, \
                                    param_grid=grid, \
                                    cv=5, scoring='accuracy')
gscv.fit(X,y)
results = pd.DataFrame(gscv.cv_results_)
results.sort_values('rank_test_score', ascending=True).head(10)

#Conduct a random search.
from scipy import stats
max_features = X.shape[1]
param_dist = {'criterion': ['gini', 'entropy'],\
              'max_features': stats.randint(low=1, \
                                            high=max_features)}
rscv = model_selection.RandomizedSearchCV\
       (estimator=rfc, param_distributions=param_dist,\
        n_iter=50, cv=5, scoring='accuracy', random_state=100)
rscv.fit(X,y)
results = pd.DataFrame(rscv.cv_results_)
results.sort_values('rank_test_score', ascending=True).head(10)
```

A truncated output is as follows:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_criterion	param_max_features	params	split0_test_score
6	1.278512	0.134416	0.018406	0.000558	gini	99	{'criterion': 'gini', 'max_features': 99}	1.000000
25	1.347692	0.148857	0.018493	0.000672	gini	106	{'criterion': 'gini', 'max_features': 106}	1.000000
29	1.411754	0.175865	0.018339	0.000839	gini	111	{'criterion': 'gini', 'max_features': 111}	1.000000
41	1.343614	0.155845	0.018289	0.000604	gini	105	{'criterion': 'gini', 'max_features': 105}	1.000000
10	1.375382	0.161429	0.018484	0.000588	gini	108	{'criterion': 'gini', 'max_features': 108}	1.000000
18	0.307170	0.007037	0.022375	0.001073	gini	5	{'criterion': 'gini', 'max_features': 5}	0.842462

Figure 8.29: Results table output

5. Plot the mean test score versus hyperparameterization for the top 10 models found using random search:

```
results.loc[:,'params'] = results.loc[:,'params'].astype(str)
(results.sort_values('rank_test_score', ascending=False)\
        .loc[:,['params','mean_test_score']]\
        .drop_duplicates()\
        .head(10)\
        .plot.barh(x='params', xlim=(0.8)))
```

You should see a plot similar to the following:

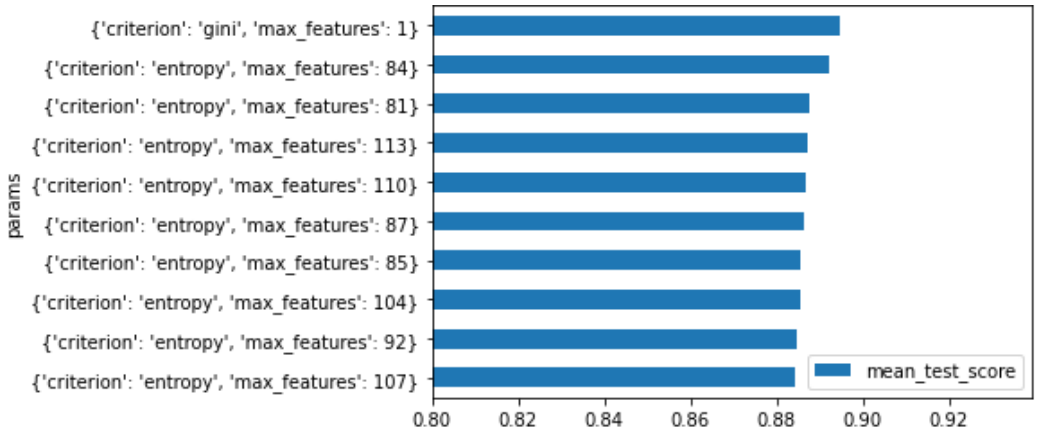

Figure 8.30: Mean test score plot

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3aJ6InS.
>
> You can also run this example online at https://packt.live/2YhA3AO.

CHAPTER 9: INTERPRETING A MACHINE LEARNING MODEL

ACTIVITY 9.01: TRAIN AND ANALYZE A NETWORK INTRUSION DETECTION MODEL

SOLUTION

1. Open a new Colab notebook.
2. Import the **`pandas`**, **`RandomForestClassifier`** from **`sklearn.ensemble`**, **`train_test_split`** from **`sklearn.model_selection`**, **`accuracy_score`** from **`sklearn.metrics`**, **`plot_partial_dependence`** from **`sklearn.inspection`**, **`altair`**, and **`feature_importance_permutation`** from **`mlxtend.evaluate`** packages:

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import altair as alt
from sklearn.inspection import plot_partial_dependence
from mlxtend.evaluate import feature_importance_permutation
```

3. Create a variable containing the URL to the dataset and load it into a DataFrame called **`df`**:

```
file_url = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter09/Dataset/KDDCup99.csv'
df = pd.read_csv(file_url)
```

4. Display the first five rows of the DataFrame using **`.head()`**:

```
df.head()
```

You should get the following output:

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	lnum_compromised	lroot_shell	lsu_attempted	lnum_root
0	0	tcp	http	SF	181	5450	0	0	0	0	0	1	0	0	0	0
1	0	tcp	http	SF	239	486	0	0	0	0	0	1	0	0	0	0
2	0	tcp	http	SF	235	1337	0	0	0	0	0	1	0	0	0	0
3	0	tcp	http	SF	219	1337	0	0	0	0	0	1	0	0	0	0
4	0	tcp	http	SF	217	2032	0	0	0	0	0	1	0	0	0	0

Figure 9.43: First five rows of df

This dataset contains numerical and categorical variables that will need to be one-hot encoded.

5. Extract the target variable, **`label`**, using **`.pop()`** and save it into a variable called **`y`**:

```
y = df.pop('label')
```

6. Perform one-hot encoding on the categorical variables using **`pd.get_dummies()`**:

```
df = pd.get_dummies(df)
```

7. Split the data into training and testing sets using **`train_test_split()`** with **`test_size=0.3`** and **`random_state=1`**:

```
X_train, X_test, y_train, y_test = train_test_split\
                                   (df, y, test_size=0.3, \
                                    random_state=1)
```

8. Instantiate **`RandomForestClassifier`** with **`random_state=168`** and fit it with the training set using **`.fit()`**:

```
rf_model = RandomForestClassifier(random_state=168)
rf_model.fit(X_train, y_train)
```

You should get the following output:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=168,
                       verbose=0, warm_start=False)
```

Figure 9.44: Logs of RandomForest

9. Extract the predictions from the training and testing sets using **`.predict()`** and save the results into two new variables – **`train_preds`** and **`test_preds`**:

```
train_preds = rf_model.predict(X_train)
test_preds = rf_model.predict(X_test)
```

10. Calculate the accuracy score for the training and testing sets using **`accuracy_score`**, save the results into two new variables, **`train_acc`** and **`test_acc`**, and print their values:

```
train_acc = accuracy_score(y_train, train_preds)
test_acc = accuracy_score(y_test, test_preds)
print(train_acc)
print(test_acc)
```

You should get something like the following output:

```
0.9999855413603845
0.9996896212029203
```

Figure 9.45: Accuracy score for the training and testing sets

The accuracy score is very similar for the training set and the testing set, so the model is not overfitting. The model achieved a very high performance of 0.999.

11. Extract the feature importance via permutation using **`feature_importance_permutation`** with the **`RandomForest`** model, the training set, **`metric='accuracy'`**, **`num_rounds=1`**, and **`seed=2`**. Print its values:

```
imp_vals, _ = feature_importance_permutation\
              (predict_method=rf_model.predict, \
               X=X_train.values, y=y_train.values, \
               metric='accuracy', num_rounds=1, seed=2)
imp_vals
```

You should get the following output:

```
array([3.18090072e-05, 7.08675762e-02, 1.73503675e-04, 0.00000000e+00,
       2.02420955e-05, 0.00000000e+00, 2.31338234e-04, 0.00000000e+00,
       1.04102205e-04, 1.44586396e-05, 8.67518377e-06, 0.00000000e+00,
       8.67518377e-06, 2.89172792e-05, 8.67518377e-06, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 8.67518377e-06, 6.94014702e-05,
       9.54270215e-05, 4.62676468e-05, 0.00000000e+00, 2.89172792e-06,
       2.89172792e-06, 1.35911212e-04, 3.84599814e-04, 1.01210477e-04,
       1.78708786e-03, 4.19300549e-04, 4.48217828e-04, 8.38601098e-05,
       9.85790049e-03, 1.49213161e-03, 3.75924630e-05, 1.73503675e-05,
       3.75924630e-05, 1.44586396e-05, 1.90854043e-04, 8.67518377e-05,
       5.78345585e-06, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 2.89172792e-06, 2.89172792e-06,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 1.44586396e-05,
       6.70880878e-04, 0.00000000e+00, 0.00000000e+00, 1.82178859e-04,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 1.15669117e-05, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 1.73503675e-04, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 1.73503675e-05, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 5.78345585e-06,
       0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
       3.18090072e-05, 0.00000000e+00])
```

Figure 9.46: Raw values for feature importance

It is quite hard to analyze the raw values of feature importance. Let's look at the top ones and plot them on a bar chart.

12. Create a DataFrame called **`perm_varimp_df`** with two columns: one called **`feature`** with the column name **`X_train`** and another one called **`importance`** containing the values of feature importance:

```
perm_varimp_df = pd.DataFrame({'feature': X_train.columns, \
                               'importance': imp_vals})
```

13. Sort the **perm_varimp_df** DataFrame on the **importance** column from the highest to the lowest value with **.sort_values()**, save the results back into the same DataFrame, and print its first five rows:

```
perm_varimp_df.sort_values('importance', ascending=False, \
                           inplace=True)
perm_varimp_df.head()
```

You should get something similar to the following output:

	feature	importance
33	dst_host_srv_diff_host_rate	0.001431
1	src_bytes	0.000807
6	hot	0.000315
29	dst_host_srv_count	0.000252
32	dst_host_same_src_port_rate	0.000121

Figure 9.47: First five rows of perm_varimp_df

The **src_bytes** variable is the most important feature for this model by far. Its value is 70 times bigger than the second most important feature.

14. Plot a **barchart** with the top 20 most important features using **perm_varimp_df**:

```
alt.Chart(perm_varimp_df[:20]).mark_bar()\
   .encode(x='importance',\
           y=alt.Y('feature:N', \
                   sort=alt.SortField(field='importance', \
                                      order='descending')))
```

You should get the following output:

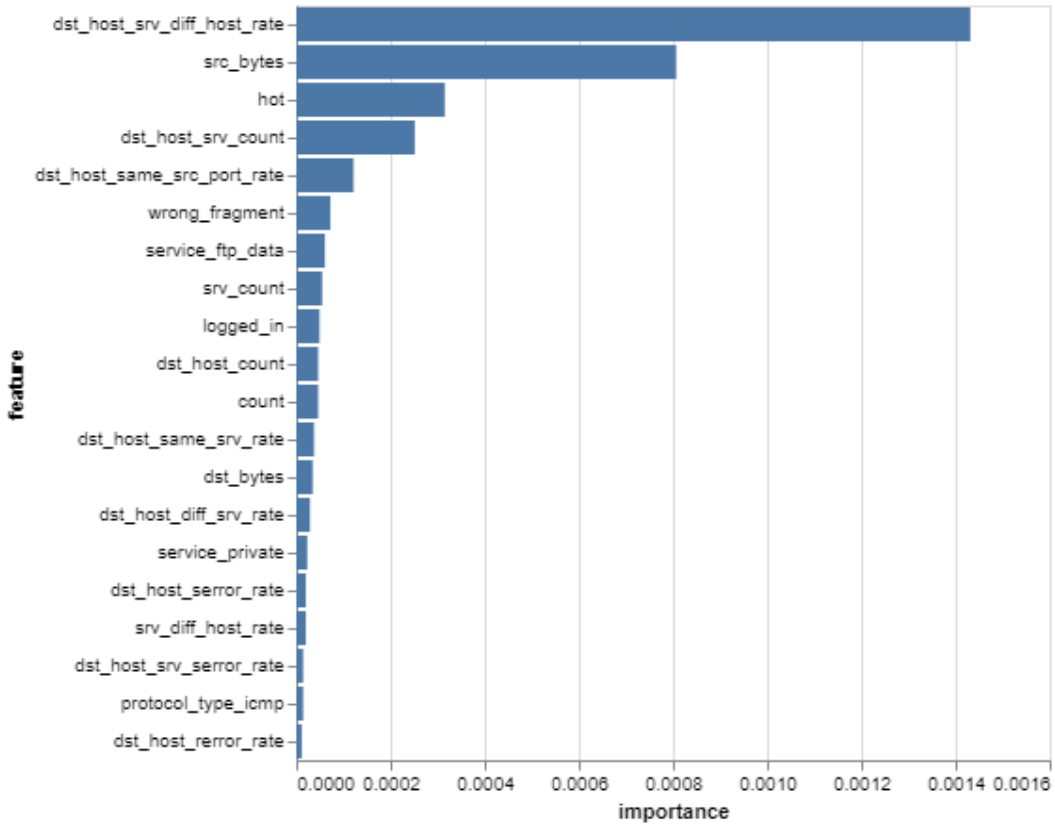

Figure 9.48: Plot of the top 20 most important features by permutation

The bar chart highlights the fact **`src_bytes`** is the most important feature by far.

15. Locate the index of the **`src_bytes`** column using **`.columns.get_loc()`** and save it into **`feature_index`**:

```
feature_index = df.columns.get_loc("src_bytes")
```

16. Display the partial dependence plot using plot_partial_dependence() with the RandomForest, the training set, feature index, the names of the features, **`target="normal"`**, **`response_method="predict_proba"`**, **`n_jobs=-1`**:

```
plot_partial_dependence(rf_model, X_train, \
                        features=[feature_index], \
                        feature_names=X_train.columns, \
                        target="normal", \
```

```
                                    response_method="predict_proba", \
                                    n_jobs=-1)
```

You should get the following output:

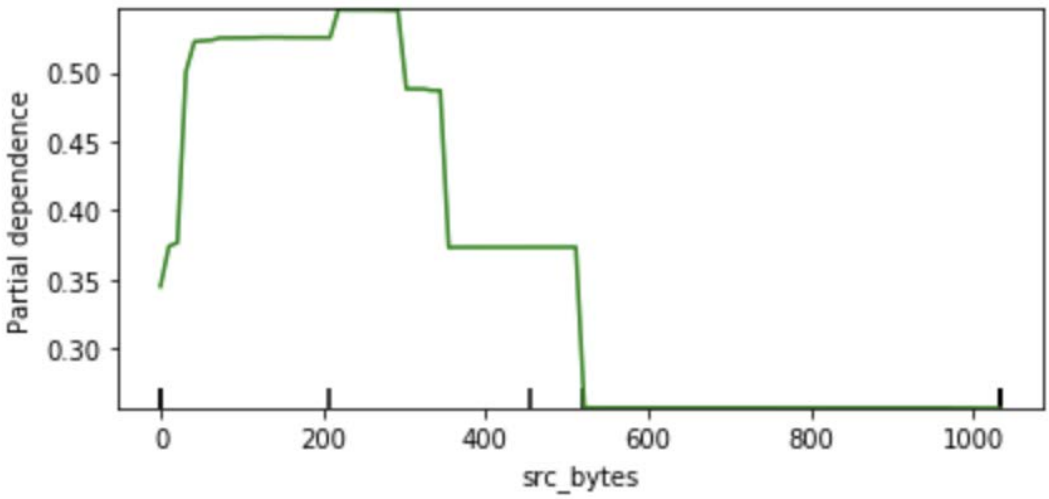

Figure 9.49: Partial dependence plot of the src_bytes feature on the normal class

This partial dependence plot shows that from 0 to 200, the probability of predicting the **`normal`** class steadily increases from 0.35 to 0.6 for **`src_bytes`**. Over 200, the probability of predicting **`normal`** decreases as **`src_bytes`** increases.

17. Install the **`lime`** package using **`!pip install`**:

```
!pip install lime
```

18. Import **`LimeTabularExplainer`** from **`lime.lime_tabular`**:

```
from lime.lime_tabular import LimeTabularExplainer
```

19. Create a new variable called **`class_names`** with the sorted list of values from **`y`**:

```
class_names = sorted(y.unique())
class_names
```

You should get the following output:

```
['back',
 'buffer_overflow',
 'ftp_write',
 'guess_passwd',
 'imap',
 'ipsweep',
 'land',
 'loadmodule',
 'multihop',
 'neptune',
 'nmap',
 'normal',
 'perl',
 'phf',
 'pod',
 'portsweep',
 'rootkit',
 'satan',
 'smurf',
 'spy',
 'teardrop',
 'warezclient',
 'warezmaster']
```

Figure 9.50: Sorted values of the target variable, y

20. Instantiate **`LimeTabularExplainer()`** with the training set, its column names, and **`class_names=class_names`**, **`mode='classification'`**:

```
lime_explainer = LimeTabularExplainer\
                 (X_train.values, \
                  feature_names=X_train.columns, \
                  class_names=class_names, \
                  mode='classification')
```

21. Display the LIME analysis on row **99893** of the testing set using **.explain_instance()** and **.show_in_notebook()**:

```
exp = lime_explainer.explain_instance(X_test.iloc[99893,], \
                                      rf_model.predict_proba, \
                                      num_features=50, \
                                      top_labels=1)
exp.show_in_notebook()
```

You should get the following output:

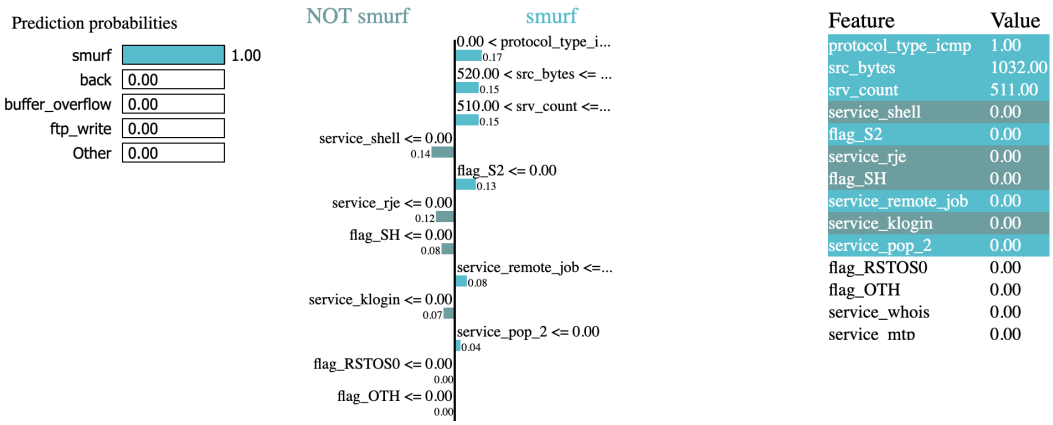

Figure 9.51: LIME analysis

The LIME output shows us that the prediction for observation **99893** is the **smurf** class. The main features that influenced this outcome were **protocol_type_icmp**, **src_bytes**, and **srv_count**.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3j1QkC9.
>
> This section does not currently have an online interactive example. You can try this code on Google Colab

CHAPTER 10: ANALYZING A DATASET

ACTIVITY 10.01: ANALYZING CHURN DATA USING VISUAL DATA ANALYSIS TECHNIQUES

SOLUTION

1. Open a new Colab notebook.
2. Import the **pandas** and **altair** packages:

```
import pandas as pd
import altair as alt
```

3. Assign the link to the churn dataset to a variable called '**file_url**' (https://packt.live/38uAquz):

```
file_url = 'https://raw.githubusercontent.com'\
           '/TrainingByPackt/The-Data-Science-Workshop'\
           '/master/Chapter10/dataset/churn.csv'
```

4. Using the **read_csv** method from the **pandas** package, load the dataset into a new variable called **'df'**:

```
df = pd.read_csv(file_url)
```

5. Print the number of rows and columns of the DataFrame using the **shape** attribute from the **pandas** package:

```
df.shape
```

You should get the following output:

```
(5000, 18)
```

6. Print out the type of each variable contained in this DataFrame using the **dtypes** attribute from the **pandas** package:

```
df.dtypes
```

You should get the following output:

```
churn                          object
accountlength                   int64
internationalplan             object
voicemailplan                 object
numbervmailmessages             int64
totaldayminutes               float64
totaldaycalls                   int64
totaldaycharge                float64
totaleveminutes               float64
totalevecalls                   int64
totalevecharge                float64
totalnightminutes             float64
totalnightcalls                 int64
totalnightcharge              float64
totalintlminutes              float64
totalintlcalls                  int64
totalintlcharge               float64
numbercustomerservicecalls      int64
dtype: object
```

Figure 10.51: List of columns and their types from the housing dataset

Most of the variables are numerical except for three: **churn**, **internationalplan**, and **voicemailplan**. Looking at the names of the numerical columns, it seems there are mostly count-like numbers of calls or minutes and monetary figures (charges). We can also note that there are a group of variables starting with **totalday**, **totaleve**, and **totalintl**, which probably means there are specific charges depending on whether the calls are made during the daytime, evening, or to overseas contacts.

7. Display the first top five rows of the DataFrame using the **head()** method from **pandas**:

```
df.head()
```

A truncated version of the output is shown below:

	churn	accountlength	internationalplan	voicemailplan	numbervmailmessages
0	No	128	no	yes	25
1	No	107	no	yes	26
2	No	137	no	no	0
3	No	84	yes	no	0
4	No	75	yes	no	0

Figure 10.52: First five rows in the churn dataset

It looks like the three categorical variables are actually binary. They can only take two values: yes or no. The **churn** column indicates whether a customer churned while **internationalplan** and **voicemailplan** indicate whether the customers subscribed to these two options.

8. Display the last five rows of the DataFrame using the **tail()** method from **pandas**:

```
df.tail()
```

You should get the following output:

	churn	accountlength	internationalplan	voicemailplan	numbervmailmessages
4995	No	50	no	yes	40
4996	Yes	152	no	no	0
4997	No	61	no	no	0
4998	No	109	no	no	0
4999	No	86	no	yes	34

Figure 10.53: Last five rows in the churn dataset

9. Display five randomly sampled rows of the DataFrame using the **`sample()`** method from **`pandas`** and pass it a **`random_state`** of **8**:

```
df.sample(n=5, random_state=8)
```

You should get the following output:

	churn	accountlength	internationalplan	voicemailplan	numbervmailmessages
2735	Yes	90	yes	yes	26
4027	Yes	67	no	no	0
1648	No	102	no	no	0
1746	Yes	60	no	no	0
4640	Yes	110	no	no	0

Figure 10.54: Five randomly sampled rows from the churn dataset

10. Use the **`describe()`** method to display some descriptive statistics for numerical variables:

```
df.describe()
```

You should get the following output:

	accountlength	numbervmailmessages	totaldayminutes	totaldaycalls	totaldaycharge
count	5000.00000	5000.000000	5000.000000	5000.000000	5000.000000
mean	100.25860	7.755200	180.288900	100.029400	30.649668
std	39.69456	13.546393	53.894699	19.831197	9.162069
min	1.00000	0.000000	0.000000	0.000000	0.000000
25%	73.00000	0.000000	143.700000	87.000000	24.430000
50%	100.00000	0.000000	180.100000	100.000000	30.620000
75%	127.00000	17.000000	216.200000	113.000000	36.750000
max	243.00000	52.000000	351.500000	165.000000	59.760000

Figure 10.55: Output of the describe() method

None of the numerical variables have negative values, which is expected as these columns contain count and spend amounts.

11. Display the list of unique values for the **churn**, **internationalplan**, and **voicemailplan** columns using a **for** loop:

```
for col_name in ['churn', 'internationalplan', 'voicemailplan']:
    print(f"{col_name}: {df[col_name].unique()}")
```

You should get the following output:

```
churn: ['No' 'Yes']
internationalplan: ['no' 'yes']
voicemailplan: ['yes' 'no']
```

Figure 10.56: List of unique values for the three categorical variables

We can confirm that the three categorical variables are binary. They can have two possible values: **yes** or **no**.

12. Create a bar plot using the **mark_bar()** and **encode()** methods from **altair** to show the frequency of each value of the 'churn' column and store it into a variable called chart1:

```
chart1 = alt.Chart(df).mark_bar()\
            .encode(alt.X("churn"), y='count()')
```

13. Create a similar bar plot for the **internationalplan** column and store it in a variable called **chart2**:

```
chart2 = alt.Chart(df).mark_bar()\
            .encode(alt.X("internationalplan"), y='count()')
```

14. Create a similar bar plot for the **'internationalplan'** column and store it in a variable called **chart2**:

```
chart3 = alt.Chart(df).mark_bar()\
            .encode(alt.X("voicemailplan"), y='count()')
```

15. Use the following syntax from **altair** to concatenate the three created charts horizontally:

```
chart1 | chart2 | chart3
```

You should get the following output:

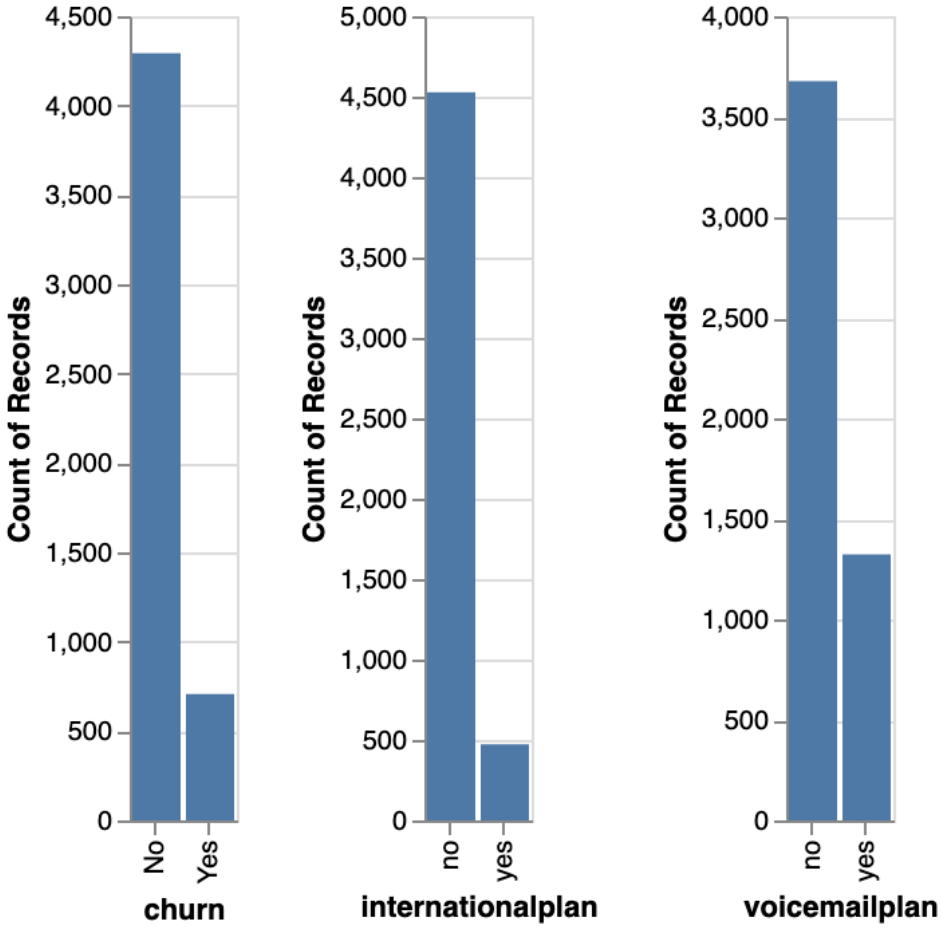

Figure 10.57: Bar charts showing the frequencies for each value contained in the three categorical variables

These three variables are imbalanced: the **`yes`** value represents between **`10%`** to **`27%`** of the data. In a real project, when predicting churn, you will have to deal with this situation by either choosing an algorithm that handles imbalanced data or resample the data to get a 50-50% split (resampling methods will be presented in *Chapter 13, Imbalanced Datasets*).

16. Create a new DataFrame called **num_df** with only the columns that are of numerical types using the **select_dtypes** method from **pandas** packages and pass in the **'number'** value to the **'include'** parameter:

```
num_df = df.select_dtypes(include='number')
```

17. Using the **columns** attribute from **pandas**, extract the list of columns of this DataFrame, '**num_df**', assign it to a new variable called '**num_cols**', and print its content:

```
num_cols = num_df.columns
num_cols.values
```

You should get the following output:

```
array(['accountlength', 'numbervmailmessages', 'totaldayminutes',
       'totaldaycalls', 'totaldaycharge', 'totaleveminutes',
       'totalevecalls', 'totalevecharge', 'totalnightminutes',
       'totalnightcalls', 'totalnightcharge', 'totalintlminutes',
       'totalintlcalls', 'totalintlcharge', 'numbercustomerservicecalls'],
      dtype=object)
```

Figure 10.58: List of numerical variables

18. Create a histogram for each numerical variable with the **mark_bar()** and **encode()** methods from the **altair** package. Use the following parameters for **alt.X**: **alt.repeat("column")**, **type='quantitative'**, and **bin=True**. Store the result in a variable called charts:

```
charts = alt.Chart(df).mark_bar()\
            .encode(alt.X(alt.repeat("column"), \
                    type='quantitative', bin=True), y='count()')
```

19. Plot the histograms using the **`repeat()`** method with the parameter **`column=list(num_cols)`**:

```
charts.repeat(column=list(num_cols))
```

You should get a series of charts in your output. The first two are shown here:

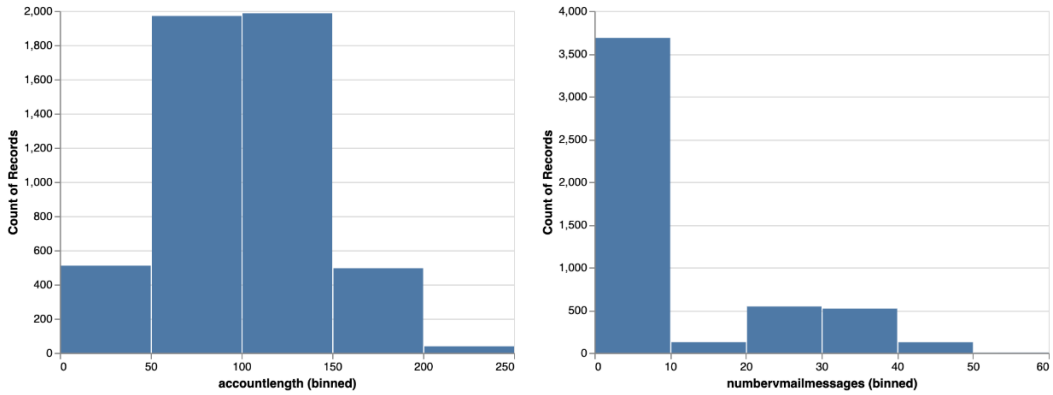

Figure 10.59: Histograms of the numerical variables

With this histogram, we can see that the number of calls that were and their duration during the day and evening are quite similar. On average, customers are making 100 calls during the day for a total of 150 minutes and another 100 calls during the evening for a total of 200 minutes. The total charges for evening calls are much cheaper than day ones: $17 and $30, respectively. We can also see that most of the customers don't make international calls: four on average.

20. Display a boxplot with **`'churn'`** as the x-axis and **`'totaldaycalls'`** as the y-axis using the **`mark_boxplot()`** method:

```
alt.Chart(df).mark_boxplot().encode(x='churn', \
                                    y='totaldaycalls')
```

You should get the following output:

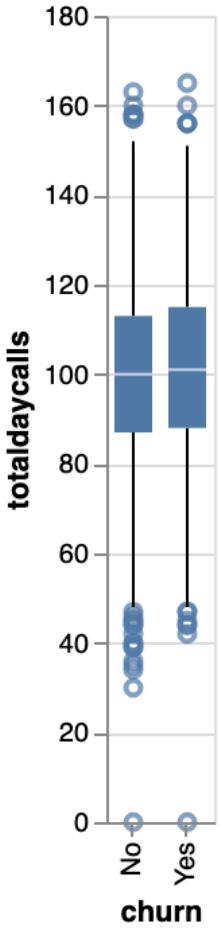

Figure 10.60: Boxplot of the totaldaycalls and churn columns

Looking at the boxplot for **`totaldaycalls`**, we can't see much difference between the group of customers who churned and the ones who didn't. The median is around 100 calls and even the IQR is very similar between **`90`** and **`110`**. This variable may not be significant in predicting churn.

21. Display a boxplot with **`churn`** as the x-axis and **`numbervmailmessages`** as the y-axis using the **`mark_boxplot()`** method:

```
alt.Chart(df).mark_boxplot().encode(x='churn',\
                                    y='numbervmailmessages')
```

You should get the following output:

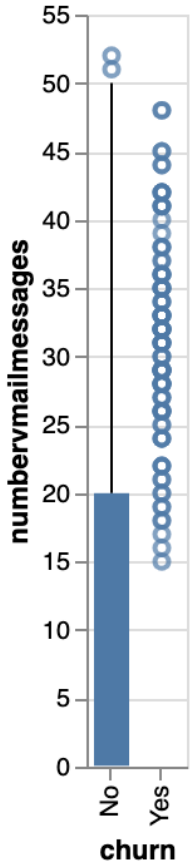

Figure 10.61: Boxplot of the numbervmailmessages and churn columns

This boxplot is showing that the distribution of the **`numbervmailmessages`** variable is very different for customers who churned or not. It seems that people that do churn tend either not to make any at all or a lot of vmail messages (over 15). This is definitely an interesting variable to feed into a predictive model.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2Qaxex6.
>
> This section does not currently have an online interactive example. You can try this code on Google Colab

CHAPTER 11: DATA PREPARATION

ACTIVITY 11.01: PREPARING THE SPEED DATING DATASET

SOLUTION

1. Open a new Colab notebook.
2. Import the **pandas** package:

```
import pandas as pd
```

3. Assign the link to the dataset to a variable called **file_url**:

```
file_url = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter11/dataset'\
           '/Speed_Dating_Data.csv'
```

Using the **read_csv()** method from the **pandas** package, load the dataset into a new variable called **df**:

```
df = pd.read_csv(file_url)
```

4. Print the first five rows of the DataFrame using the **.head()** method:

```
df.head()
```

The output below is truncated:

	iid	id	gender	idg	condtn	wave	round	position	positin1
0	1	1.0	0	1	1	1	10	7	NaN
1	1	1.0	0	1	1	1	10	7	NaN
2	1	1.0	0	1	1	1	10	7	NaN
3	1	1.0	0	1	1	1	10	7	NaN
4	1	1.0	0	1	1	1	10	7	NaN

Figure 11.56: The first five rows of the Speed Dating dataset

5. Print out the shape of the DataFrame (the number of rows and columns) using the pandas **.shape** attribute:

```
df.shape
```

You should get the following output:

```
(8378, 195)
```

This dataset contains quite a lot of features (**195**) for **8378** rows. Let's check whether there are any duplicate rows in it.

6. Print out the number of duplicate rows (looking at all the columns in the DataFrame) by combining the pandas **.duplicated()** and **.sum()** methods:

```
df.duplicated().sum()
```

You should get the following output:

```
0
```

By looking at the 195 columns of this dataset, we can see that there are no duplicate rows at all, which is great. Now, let's look at the identifier variables listed in the dataset description document.

7. Print out the number of duplicate rows, like we did in *step 6*, but this time only look at the identifier columns (**'iid'**,**'id'**,**'partner'**, and **'pid'**). You can do this by specifying the **subset** parameter:

```
df.duplicated(subset=['iid','id','partner','pid']).sum()
```

You should get the following output:

```
0
```

As we can see, there are no duplicate rows in this dataset.

If you looked at the dataset description document (https://packt.live/2Qrp7gD), then you'll know that the values of the following variables should range between 1 and 10: **'imprace'**, **'imprelig'**, **'sports'**, **'tvsports'**, **'exercise'**, **'dining'**, **'museums'**, **'art'**, **'hiking'**, **'gaming'**, **'clubbing'**, **'reading'**, **'tv'**, **'theater'**, **'movies'**, **'concerts'**, **'music'**, **'shopping'**, **'yoga'**, **'exphappy'**, and **'satis_2'**. In the next few steps, you are going to check that there are no unexpected values in these columns.

8. Create a variable called scale_1_10. This will list the following column names: **'imprace'**, **'imprelig'**, **'sports'**, **'tvsports'**, **'exercise'**, **'dining'**, **'museums'**, **'art'**, **'hiking'**, **'gaming'**, **'clubbing'**, **'reading'**, **'tv'**, **'theater'**, **'movies'**, **'concerts'**, **'music'**, **'shopping'**, **'yoga'**, **'exphappy'**, and **'satis_2'**:

```
scale_1_10 = ['imprace', 'imprelig', 'sports', \
              'tvsports', 'exercise', 'dining', \
              'museums', 'art', 'hiking', 'gaming', \
              'clubbing', 'reading', 'tv', 'theater', \
              'movies', 'concerts', 'music', \
              'shopping', 'yoga', 'exphappy', 'satis_2']
```

9. Create a function called **check_range** that takes a column (a **pandas** series), a minimum value, and a maximum value as input parameters. The function will check whether each row of the given column is outside the given range (below the minimum value or above the maximum value) and return the corresponding list of binary values:

```
def check_range(column, min_value, max_value):
    return (column < min_value) | (column >max_value)
```

10. Test your function on the **imprace** column with **1** and **10** as the minimum and maximum values, respectively. Then, save the output into a variable called **unexpected_mask** and print the **sum** to check how many cases are outside this range:

```
unexpected_mask = check_range(df['imprace'], 1, 10)
unexpected_mask.sum()
```

You should get the following output:

```
8
```

Here, there are **8** rows that have values for **'imprace'** outside the expected range (between 1 and 10).

11. Define a function called **`print_unexpected`** that takes a DataFrame, a column name, and a list of binary values as input parameters. This function will check whether the sum of the binary values is over 0. If it is, the case prints out the column name, this sum, and the unique values of the given columns and the rows that match the binary list (keeping only **`True`** values) using the pandas **`.loc`** and **`.unique()`** methods:

```
def print_unexpected(df, col_name, unexpected_mask):
    if unexpected_mask.sum() > 0:
        print(col_name)
        print(unexpected_mask.sum())
        print(df.loc[unexpected_mask,col_name].unique())
```

12. Test your function on the **`'imprace'`** column using the output of the previous function, that is, **`unexpected_mask`**:

```
print_unexpected(df, 'imprace', unexpected_mask)
```

You should get the following output:

```
imprace
8
[0.]
```

Figure 11.57: Number of rows with unexpected values for 'imprace' and a list of unexpected values

As you can see, you still have **8** cases that are outside the expected range for this column and that the unexpected value is **0**.

13. Create a function called **`check_ranges`** that takes a DataFrame, a list of columns, and minimum and maximum values as input parameters. This function will iterate through each column from the given column list, call the **`check_range`** function, and pass its output to the **`print_unexpected`** function, which you defined in *steps 10* and *12*:

```
def check_ranges(df, col_list, min_value, max_value):
    for col_name in col_list:
        unexpected_mask = check_range(df[col_name], \
                                      min_value, max_value)
        print_unexpected(df, col_name, unexpected_mask)
```

14. Test this function with the dataset and the **`scale_1_10`** variables you defined in *step 9* , and use **`1`** and **`10`** as their minimum and maximum values, respectively:

```
check_ranges(df, scale_1_10, 1, 10)
```

The output below is truncated:

```
imprace
8
[0.]
museums
18
[0.]
art
18
[0.]
```

Figure 11.58: Number of rows with unexpected values and a list of unexpected values for each column

As you can see, most of these columns have the unexpected value **0**, while some of them have values of 13 and 14. In a real project, you will probably go and ask the surveyors if these values are expected or not. Let's say they confirmed that the value **0** is a possible value in the survey, but not 13 and 14; they think this was just an error that occurred while they recorded these cases and that the values should be 10. Let's look at how we would fix such issues.

Create a function called **`replace_value`** that takes a DataFrame, a column name, an incorrect value, and a new value as input parameters. This function will subset all the rows that are equal to the incorrect value for the given column and replace it with the new given value. Then, it will print out the column's name and the list of unique values for this column:

```
def replace_value(df, col_name, incorrect_value, new_value):
    df.loc[df[col_name] == incorrect_value, \
                           col_name] = new_value
    print(col_name)
    print(df[col_name].unique())
```

15. Test the **replace_value** function on the **gaming** column, where **14** is the incorrect value and **10** is the new value:

```
replace_value(df, 'gaming', 14, 10)
```

You should get the following output:

```
gaming
[ 1.  5.  4.  6.  2.  3.  7.  8. 10. nan  9.  0.]
```

Figure 11.59: List of unique values for 'gaming'

Now that we've replaced **14**, it is no longer one of the possible values for this column.

16. Use the **replace_value** function on the **reading** column, where **13** is the incorrect value and **10** is the new value:

```
replace_value(df, 'reading', 13, 10)
```

You should get the following output:

```
reading
[ 6. 10.  7.  9.  8.  4.  5. nan  2.  3.  1.]
```

Figure 11.60: List of unique values for 'reading'

Now that we've replaced **13**, it is no longer one of the possible values for this column.

17. Create a **for** loop that will iterate through the following suffixes: **['1_1', '1_2', '1_3', '1_s', '2_1', '2_2', '2_3', '4_1', '4_2', '4_3', '7_2',** and **'7_3']**. For each of them, create a list comprehension (or another for loop) so that you can extract the columns that contain the given suffix by using the **.endswith()** method and store them into a variable called **suffix_cols**. Then, apply the **check_ranges** function to this list and use **0** and **100** as their minimum and maximum values, respectively:

```
for suffix in ['1_1', '1_2', '1_3', '1_s', '2_1', \
               '2_2', '2_3', '4_1', '4_2', '4_3', \
               '7_2', '7_3']:
    suffix_cols = [col for col in df.columns if \
                   col.endswith(suffix)]
    check_ranges(df, suffix_cols, 0, 100)
```

No output is displayed, which means that all these columns have values within the expected range, that is, between 0 and 100.

18. Create a for loop that's similar to what we created in *step 19* for the following suffixes, where **1** and **10** are the minimum and maximum values, respectively: **['3_1', '3_2', '3_3', '5_1', '5_2', '5_3', '3_s']**:

```
for suffix in ['3_1', '3_2', '3_3', '5_1', '5_2', \
               '5_3', '3_s']:
    suffix_cols = [col for col in df.columns if \
                   col.endswith(suffix)]
    check_ranges(df, suffix_cols, 1, 10)
```

You should get the following output:

```
attr3_3
112
[12.]
sinc3_3
173
[12.]
intel3_3
233
[12.]
fun3_3
153
[12.]
amb3_3
147
[12.]
```

Figure 11.61: Number of rows with unexpected values and a list of unexpected values for each column

As you can see, all the columns ending with **3_3** have **12** as their unexpected values. Let's say that, after consultation with the surveyors, you agree to replace these values with **10**.

19. Create a **for** loop that iterates through the list of columns ending with **3_3** and call the **replace_values** function for each of them. Provide **12** as the incorrect value and **10** as the new value:

```
for col_name in ['attr3_3', 'sinc3_3', 'intel3_3', \
                 'fun3_3', 'amb3_3']:
    replace_value(df, col_name, 12, 10)
```

You should get the following output:

```
attr3_3
[ 5.  7. nan  6.  4.  9.  8.  3. 10.  2.]
sinc3_3
[ 7.  6. nan  5.  8.  9. 10.  4.  3.  2.]
intel3_3
[ 7.  9. nan  6. 10.  8.  5.  4.  3.]
fun3_3
[ 7.  9. nan  8.  6.  3.  5. 10.  2.  4.]
amb3_3
[ 7.  4. nan  5. 10.  9.  8.  6.  2.  3.  1.]
```

Figure 11.62: List of unique values for variables ending with 3_3

You have fixed the unexpected values for these columns.

20. Print the data type of each variable using the **dtypes** attribute:

```
df.dtypes
```

You should get the following (truncated) output:

```
iid           int64
id          float64
gender        int64
idg           int64
condtn        int64
wave          int64
round         int64
```

Figure 11.63: Data types of each column

As you can see, most of the columns have been detected as numerical variables, but by looking at the dataset description document, you know that most of them are categorical. Let's change their data types.

21. Create a list called **num_cols** that contains the following list of columns: **'round'**, **'order'**, **'int_corr'**, **'age'**, **'mn_sat'**, **'income'**, and **'expnum'**:

```
num_cols = ['round', 'order', 'int_corr', 'age', \
            'mn_sat', 'income', 'expnum']
```

22. Create another list called **cat_cols** that contains the remaining column names (excluding the ones in **num_cols**) of this DataFrame using the attribute columns combined with the **.difference()** method:

```
cat_cols = df.columns.difference(num_cols)
```

23. Create a **for** loop that will iterate through **cat_cols** and change the data type for each column into a category by using the **.astype()** method:

```
for col_name in cat_cols:
    df[col_name] = df[col_name].astype('category')
```

24. Print the data type of each variable using the **dtypes** attribute:

```
df.dtypes
```

You should get the following (truncated) output:

```
iid          category
id           category
gender       category
idg          category
condtn       category
wave         category
round           int64
position     category
```

Figure 11.64: Data types of each column

You have sorted out the data types for each column. Now, let's see whether we have any missing columns in the numerical fields.

25. Print the number of missing values for each column in **num_cols** by combining the **.isna()** and **.sum()** methods:

```
df[num_cols].isna().sum()
```

You should get the following output:

```
round          0
order          0
int_corr     158
age           95
mn_sat      5245
income      4099
expnum      6578
dtype: int64
```

Figure 11.65: Number of missing values for numerical variables

There are some missing values for most of these columns. You need to fix these cases. Let's start with the **'int_corr'** column.

26. Print the unique values of the **'int_corr'** variable using the **.unique()** method:

```
df['int_corr'].unique()
```

You should get the following (truncated) output:

```
array([ 0.14,  0.54,  0.16,  0.61,  0.21,  0.25,  0.34,  0.5 ,  0.28,
       -0.36,  0.29,  0.18,  0.1 , -0.21,  0.32,  0.73,  0.6 ,  0.07,
        0.11,  0.39, -0.24, -0.14,  0.09, -0.04, -0.3 , -0.26, -0.15,
       -0.47, -0.18,  0.05,  0.37,  0.35,  0.15, -0.19, -0.43,  0.  ,
       -0.17,  0.08, -0.16,  0.06, -0.05, -0.13, -0.06,  0.33, -0.51,
        0.12,  0.19,  0.47,  0.03,  0.46,  0.43,  0.52, -0.46, -0.27,
        0.59,  0.31, -0.34, -0.03, -0.11,  0.42, -0.4 , -0.23,  0.17,
        0.68, -0.01, -0.35,  0.3 ,  0.65,  0.24,  0.41,  0.49,  0.01,
        0.22, -0.08,  0.27,  0.44,  0.62, -0.2 , -0.02, -0.33, -0.52,
       -0.1 ,  0.58, -0.57, -0.31, -0.07, -0.32,  0.04, -0.12,  0.48,
```

Figure 11.66: List of unique values for 'int_corr'

The values of the **int_corr** column range between **-1** and **1**. It seems like they have been normalized. Since there are no extreme values or outliers, you can impute the missing values with the mean of this variable. This is what you are going to do in the next few steps.

27. Create a condition mask called **`int_corr_mask`** that finds the missing values in the **`'int_corr'`** column by using the **`.isna()`** method:

```
int_corr_mask = df['int_corr'].isna()
```

28. Display the number of missing values for this column by using the **`.sum()`** method on **`int_corr_mask`**:

```
int_corr_mask.sum()
```

You should get the following output:

```
158
```

You got the exact same number of missing values for **`int_corr`** that you got in *Step 27*.

29. Extract the mean of **`int_corr`** using the **`.mean()`** method and store it in a new variable called **`int_corr_mean`**. Print out its value:

```
int_corr_mean = df['int_corr'].mean()
print(int_corr_mean)
```

You should get the following output:

```
0.19600973236009664
```

The average value for this column is **`0.196`**. You need to replace all the missing values with this value in the **`int_corr`** column.

30. Replace all the missing values in the **`int_corr`** variable with their averages by using the **`.fillna()`** method along with the **`inplace=True`** parameter:

```
df['int_corr'].fillna(int_corr_mean, inplace=True)
```

31. Print the number of missing values for **`int_corr`** by combining the **`.isna()`** and **`.sum()`** methods:

```
df['int_corr'].isna().sum()
```

You should get the following output:

```
0
```

There are no more missing values in the variable.

32. Create a new variable called **`missing_num_cols`** that contains the following columns: **`age`**, **`mn_sat`**, **`income`**, and **`expnum`**:

```
missing_num_cols = ['age', 'mn_sat', 'income', 'expnum']
```

33. Create a **`for`** loop that will iterate through the columns in **`missing_num_cols`** and print out their names and a list of their unique values using the **`.unique()`** method:

```
for col_name in missing_num_cols:
    print(col_name)
    print(df[col_name].unique())
```

You should get the following (truncated) output:

```
age
[21. 24. 25. 23. 22. 26. 27. 30. 28. nan 29. 34. 35. 32. 39. 20. 19. 18.
 37. 33. 36. 31. 42. 38. 55.]
mn_sat
[  nan 1070. 1258. 1400. 1290. 1460. 1430. 1215. 1330. 1450. 1155. 1140.
 1360. 1402. 1250. 1210. 1220. 1410. 1260. 1380. 1030. 1309. 1308. 1050.
 1100. 1310. 1490. 1188. 1097. 1212. 1340. 1034. 1185. 1242. 1160. 1099.
 1214. 1270. 1110. 1178. 1060. 1157. 1180. 1014. 1341.  990. 1320. 1159.
 1370. 1105. 1365. 1011. 1130. 1206. 1331. 1191.  914. 1200. 1080. 1090.
 1092. 1470. 1149. 1134. 1230. 1267. 1280. 1227. 1239.]
income
```

Figure 11.67: List of unique values for numerical variables

The values for these columns haven't been normalized and some of them have **outliers**. This time, you are going to need to use their medians to fill in the missing values.

34. Create a **`for`** loop, like the one we created in *Step 35*, but this time calculate the median of each column and save it into a variable called **`col_median`**. Then, impute the missing values with this median value by using the **`.fillna()`** method along with the **`inplace=True`** parameter and print the name of the column and its median value:

```
for col_name in missing_num_cols:
    col_median = df[col_name].median()
    df[col_name].fillna(col_median, inplace=True)
    print(col_name)
    print(col_median)
```

You should get the following output:

```
age
26.0
mn_sat
1310.0
income
43185.0
expnum
4.0
```

Figure 11.68: Average values of the numerical variables

35. Create a **`for`** loop to print the name of each column and its number of missing values by combining the **`.isna()`** and **`.sum()`** methods:

```
for col_name in missing_num_cols:
    print(col_name)
    print(df[col_name].isna().sum())
```

You should get the following output:

```
age
0
mn_sat
0
income
0
expnum
0
```

Figure 11.69: Number of missing values for numerical variables

In this activity, you have cleaned up most of the main quality issues in this dataset. You looked for duplication, incorrect values, incorrect data types, and missing values. You have put all the techniques you learned about in this chapter into practice in order to fix these issues. You are now more confident in using this modified version of the dataset to build a matching algorithm if you really were to work on this project.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3heb7BL.
>
> You can also run this example online at https://packt.live/32cY5y8.

CHAPTER 12: FEATURE ENGINEERING

ACTIVITY 12.01: FEATURE ENGINEERING ON A FINANCIAL DATASET

SOLUTION

1. Open up a new Colab notebook and import the **pandas** package:

```
import pandas as pd
```

2. Assign the links to the **disp**, **trans, account**, and **client tables** from the Financial dataset to four new variables called **disp_url**, **trans_url**, **account_url**, and **client_url**:

```
disp_url = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter12/Dataset/disp.csv'
trans_url = 'https://raw.githubusercontent.com'\
            '/PacktWorkshops/The-Data-Science-Workshop'\
            '/master/Chapter12/Dataset/trans.csv'
account_url = 'https://raw.githubusercontent.com'\
              '/PacktWorkshops/The-Data-Science-Workshop'\
              '/master/Chapter12/Dataset/account.csv'
client_url = 'https://raw.githubusercontent.com'\
             '/PacktWorkshops/The-Data-Science-Workshop'\
             '/master/Chapter12/Dataset/client.csv'
```

3. Using the **read_csv()** method from the **pandas** package, load the four tables into four new variables called **df_disp**, **df_trans**, **df_account**, and **df_client**. Specify the **sep=';'** parameter as this file is not separated by commas but semi-colons:

```
df_disp = pd.read_csv(disp_url, sep=';')
df_trans = pd.read_csv(trans_url, sep=';')
df_account = pd.read_csv(account_url, sep=';')
df_client = pd.read_csv(client_url, sep=';')
```

4. Print the first five rows of **df_trans** using the **.head()** method:

```
df_trans.head()
```

You should get the following output:

	trans_id	account_id	date	type	operation	amount	balance	k_symbol	bank	account
0	695247	2378	930101	PRIJEM	VKLAD	700.0	700.0	NaN	NaN	NaN
1	171812	576	930101	PRIJEM	VKLAD	900.0	900.0	NaN	NaN	NaN
2	207264	704	930101	PRIJEM	VKLAD	1000.0	1000.0	NaN	NaN	NaN
3	1117247	3818	930101	PRIJEM	VKLAD	600.0	600.0	NaN	NaN	NaN
4	579373	1972	930102	PRIJEM	VKLAD	400.0	400.0	NaN	NaN	NaN

Figure 12.48: First five rows of df_trans

5. Print the shape of **df_trans** using the **.shape** attribute:

```
df_trans.shape
```

You should get the following output:

```
(1056320 ,10)
```

6. Print the first five rows of **df_account** using the **.head()** method:

```
df_account.head()
```

You should get the following output:

	account_id	district_id	frequency	date
0	576	55	POPLATEK MESICNE	930101
1	3818	74	POPLATEK MESICNE	930101
2	704	55	POPLATEK MESICNE	930101
3	2378	16	POPLATEK MESICNE	930101
4	2632	24	POPLATEK MESICNE	930102

Figure 12.49: First five rows of df_account

7. Merge **df_trans** and **df_account** together using left join on the **account_id** column and save the new DataFrame called **df_trans_acc**:

```
df_trans_acc = pd.merge(df_trans, df_account, \
                        how='left', on='account_id')
```

8. Print the shape of **df_trans_acc**:

```
df_trans_acc.shape
```

You should get the following output:

```
(1056320, 13)
```

9. Print the first five rows of **df_disp** using the **.head()** method:

```
df_disp.head()
```

You should get the following output:

	disp_id	client_id	account_id	type
0	1	1	1	OWNER
1	2	2	2	OWNER
2	3	3	2	DISPONENT
3	4	4	3	OWNER
4	5	5	3	DISPONENT

Figure 12.50: First five rows of df_disp

We can see that the **account_id** column doesn't contain a unique identifier, and this will add additional rows after the merge. Let's subset this DataFrame to only the **OWNER** type.

10. Subset **df_disp** to only keep the rows with **OWNER** as their type and save the results in a new DataFrame called **df_disp_owner**:

```
df_disp_owner = df_disp[df_disp['type'] == 'OWNER']
```

11. Check the number of duplicates on the **account_id** column of **df_disp_owner** using the **.duplicated()** and **.sum()** methods:

```
df_disp_owner.duplicated(subset='account_id').sum()
```

You should get the following output:

```
0
```

All account IDs are unique now. Let's merge the DataFrames.

12. Merge **df_trans_acc** and **df_disp_owner** together using left join on the **account_id** column. Save the new DataFrame called **df_trans_acc_disp** and print its shape:

```
df_trans_acc_disp = pd.merge(df_trans_acc, df_disp_owner, \
                             how='left', on='account_id')
df_trans_acc_disp.shape
```

You should get the following output:

```
(1056320, 16)
```

13. Print the first five rows of **df_client** using the **.head()** method:

```
df_client.head()
```

You should get the following output:

	client_id	birth_number	district_id
0	1	706213	18
1	2	450204	1
2	3	406009	1
3	4	561201	5
4	5	605703	5

Figure 12.51: First five rows of df_client

14. Merge **df_trans_acc_disp** and **df_client** together using left join on the **client_id** and **district_id** columns. Save the new DataFrame called **df_merged** and print its shape:

```
df_merged = pd.merge(df_trans_acc_disp, df_client, how='left', \
                     on=['client_id', 'district_id'])
df_merged.shape
```

You should get the following output:

```
(1056320, 17)
```

15. Print the column names of **df_merged** using the **.columns** attribute:

```
df_merged.columns
```

You should get the following output:

```
Index(['trans_id', 'account_id', 'date_x', 'type_x', 'operation', 'amount',
       'balance', 'k_symbol', 'bank', 'account', 'district_id', 'frequency',
       'date_y', 'disp_id', 'client_id', 'type_y', 'birth_number'],
      dtype='object')
```

Figure 12.52: Columns names of df_merged

16. Rename the **date_x**, **type_x**, **date_y** and **type_y** columns **trans_date**, **trans_type**, **account_creation** and **client_type** respectively:

```
df_merged.rename(columns={'date_x': 'trans_date', \
                          'type_x': 'trans_type', \
                          'date_y':'account_creation', \
                          'type_y':'client_type'}, \
                          inplace=True)
```

17. Print the first five rows of **df_merged** using the **.head()** method:

```
df_merged.head()
```

You should get the following (truncated) output:

account_creation	disp_id	client_id	client_type	birth_number
930101	2873	2873	OWNER	755324.0
930101	692	692	OWNER	NaN
930101	844	844	OWNER	NaN
930101	4601	4601	OWNER	NaN
930102	2397	2397	OWNER	NaN

Figure 12.53: First 5 rows of df_merged

18. Print the data types of each column using the **.dtypes** attribute:

```
df_merged.dtypes
```

You should get the following output:

```
trans_id              int64
account_id            int64
trans_date            int64
trans_type           object
operation            object
amount              float64
balance             float64
k_symbol             object
bank                 object
account             float64
district_id           int64
frequency            object
account_creation      int64
disp_id               int64
client_id             int64
client_type          object
birth_number        float64
dtype: object
```

Figure 12.54: Data types of df_merged

The **trans_date** and **account_creation** columns are integers. We need to convert them into datetime.

19. Convert the **trans_date** and **account_creation** columns using **.to_datetime()** with the **format="%y%m%d"** parameter:

```
df_merged['trans_date'] = pd.to_datetime\
                          (df_merged['trans_date'], \
                           format="%y%m%d")
df_merged['account_creation'] = pd.to_datetime\
                                (df_merged['account_creation'],\
                                 format="%y%m%d")
```

20. Print the data types of each column using the **.dtypes** attribute:

```
df_merged.dtypes
```

You should get the following output:

```
trans_id                     int64
account_id                   int64
trans_date          datetime64[ns]
trans_type                  object
operation                   object
amount                     float64
balance                    float64
k_symbol                    object
bank                        object
account                    float64
district_id                  int64
frequency                   object
account_creation    datetime64[ns]
disp_id                      int64
client_id                    int64
client_type                 object
birth_number               float64
dtype: object
```

Figure 12.55: Data types of df_merged

We need to perform some transformations on the **`birth_number`** column because it has specific code that includes the date of birth of a person and also their sex.

Here is the rule used for this code:

birthday and sex: the number is in the form YYMMDD for men, the number is in the form YYMM+50DD for women, where YYMMDD is the date of birth.

> **NOTE**
>
> More details on this code can be found here: https://packt.live/2TQoE9R

21. Create a new column called **`is_female`** by performing the following calculation to extract the sex information: **`(df_merged['birth_number'] % 10000) / 5000 > 1`**

```
df_merged['is_female'] = (df_merged['birth_number'] % 10000) \
                         / 5000 > 1
```

22. Print out the first five rows of the **`birth_number`** column:

```
df_merged['birth_number'].head()
```

You should get the following output:

```
0    755324.0
1         NaN
2         NaN
3         NaN
4         NaN
Name: birth_number, dtype: float64
```

Figure 12.56: First five rows of birth_number

23. Transform all the rows with **`is_female`** is **`True`** by removing the value within the column **`birth_number`** by **`5000`**:

```
df_merged.loc[df_merged['is_female'] == True, \
              'birth_number'] -= 5000
```

24. Print out the first five rows of the **`'birth_number'`** column:

```
df_merged['birth_number'].head()
```

You should get the following output:

```
0    750324.0
1         NaN
2         NaN
3         NaN
4         NaN
Name: birth_number, dtype: float64
```

Figure 12.57: First five rows of 'birth_number'

25. Convert the **`birth_number`** column to the **`.to_datetime()`** method with the **`format="%y%m%d"`**, **`errors='coerce'`** parameters:

```
pd.to_datetime(df_merged['birth_number'], \
               format="%y%m%d", errors='coerce')
```

You should get the following output (truncated):

```
0          1975-03-24
1                 NaT
2                 NaT
3                 NaT
4                 NaT
5          2038-08-12
6                 NaT
7          1979-03-24
8          1971-03-02
9                 NaT
10         1970-06-24
11                NaT
12                NaT
13         2028-04-02
14         2040-12-02
```

Figure 12.58: First converted rows of birth_number

Because the year was recorded with only two digits in this dataset, the date is converted to either 20*th* or 21*st* century years. We need to fix this issue.

26. Convert the **`birth_number`** column to a string using the **`.astype()`** method and print out the first five rows of the **`birth_number`** column:

```
df_merged['birth_number'] = df_merged['birth_number'].astype(str)
df_merged['birth_number'].head()
```

You should get the following output:

```
0    750324.0
1         nan
2         nan
3         nan
4         nan
Name: birth_number, dtype: object
```

Figure 12.59: First 5 rows of 'birth_number'

After the conversion to a string, all the missing values are converted to a string with the **'nan'** value. Let's convert them back to proper missing values.

27. Import the **numpy** package and change the value of **birth_number** for all rows with value **nan** to **np.nan**. Print out the first five rows of the **birth_number** column:

```
import numpy as np
df_merged.loc[df_merged['birth_number'] == 'nan', \
              'birth_number'] = np.nan
df_merged['birth_number'].head()
```

You should get the following output:

```
0    750324.0
1         NaN
2         NaN
3         NaN
4         NaN
Name: birth_number, dtype: object
```

Figure 12.60: First five rows of birth_number

28. Add the **19** prefix to **birth_number** for all rows that don't have missing values for this column:

```
df_merged.loc[~df_merged['birth_number'].isna(), \
              'birth_number'] \
= '19' + df_merged.loc[~df_merged['birth_number']\
                       .isna(), 'birth_number']
df_merged['birth_number'].head()
```

You should get the following output:

```
0    19750324.0
1           NaN
2           NaN
3           NaN
4           NaN
Name: birth_number, dtype: object
```

Figure 12.61: First five rows of 'birth_number'

29. Convert the **birth_number** column to the **.to_datetime()** method with the following parameters: **format="%y%m%d", errors='coerce'** and save the results back to **birth_number**. Print out the first 20 rows of the **birth_number** column:

```
df_merged['birth_number'] = pd.to_datetime\
                            (df_merged['birth_number'], \
                             format="%Y%m%d", errors='coerce')
df_merged['birth_number'].head(20)
```

You should get the following output (truncated):

```
0    1975-03-24
1           NaT
2           NaT
3           NaT
4           NaT
5    1938-08-12
6           NaT
7    1979-03-24
8    1971-03-02
9           NaT
10   1970-06-24
```

Figure 12.62: First 20 converted rows of birth_number

We have fixed dealing with the year issue. We can now create a new feature that will calculate the age of the customer when their account was created.

30. Create a new column called **age_at_creation** by subtracting **account_creation** from **birth_number**:

```
df_merged['age_at_creation'] = df_merged['account_creation'] \
                               - df_merged['birth_number']
```

31. Convert the **timedelta** results in **age_at_creation** by dividing them by **np.timedelta64(1,'Y')**:

```
df_merged['age_at_creation'] = df_merged['age_at_creation'] \
                               / np.timedelta64(1,'Y')
```

32. Convert **`age_at_creation`** to an integer using the **`.round()`** method. Print the first five rows of **`df_merged`**:

```
df_merged['age_at_creation'] = df_merged['age_at_creation']\
                               .round()
df_merged.head()
```

You should get the following output (truncated):

frequency	account_creation	disp_id	client_id	client_type	birth_number	is_female	age_at_creation
POPLATEK MESICNE	1993-01-01	2873	2873	OWNER	1975-03-24	True	18.0
POPLATEK MESICNE	1993-01-01	692	692	OWNER	NaT	False	NaN
POPLATEK MESICNE	1993-01-01	844	844	OWNER	NaT	False	NaN
POPLATEK MESICNE	1993-01-01	4601	4601	OWNER	NaT	False	NaN
POPLATEK MESICNE	1993-01-02	2397	2397	OWNER	NaT	False	NaN

Figure 12.63: First five rows of df_merged

In this activity, we created new features by merging different tables together and manipulating date columns. We now have a much richer dataset with extra valuable information that can be fed to a machine learning model.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3kYTzff.
>
> You can also run this example online at https://packt.live/2EcKglb.

CHAPTER 13: IMBALANCED DATASETS

ACTIVITY 13.01: FINDING THE BEST BALANCING TECHNIQUE BY FITTING A CLASSIFIER ON THE TELECOM CHURN DATASET

Solution

1. Open a Colab notebook. Install smote-variants in preparation for later steps:

```
!pip install smote-variants
```

2. Load the data from the GitHub repository. This is loaded using the following code snippet:

```
# Loading data from the Github repository
import pandas as pd
filename = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter13/Dataset/churn.csv'
# Loading the data using pandas
churnData = pd.read_csv(filename,sep=",")
churnData.head()
```

You should get the following output:

	churn	accountlength	internationalplan	voicemailplan	numbervmailmessages	totaldayminutes	totaldaycalls	totaldaycharge	totaleveminutes	totalevecalls
0	No	128	no	yes	25	265.1	110	45.07	197.4	99
1	No	107	no	yes	26	161.6	123	27.47	195.5	103
2	No	137	no	no	0	243.4	114	41.38	121.2	110
3	No	84	yes	no	0	299.4	71	50.90	61.9	88
4	No	75	yes	no	0	166.7	113	28.34	148.3	122

Figure 13.22: First five rows of the dataset

3. Normalize data using **`MinMaxScaler`** and drop the original columns that were transformed.

   In this step, all the numerical variables are transformed using the **`MinMaxScaler()`** function:

```
# Normalising data
from sklearn import preprocessing

minmaxScaler = preprocessing.MinMaxScaler()

# Converting each of the columns to scaled version
```

```
churnData['alScaled'] = minmaxScaler.fit_transform\
                        (churnData['accountlength']\
                         .values.reshape(-1,1))
churnData['nvmmScaled'] = minmaxScaler.fit_transform\
                          (churnData['numbervmailmessages']\
                           .values.reshape(-1,1))
churnData['tdmScaled'] = minmaxScaler.fit_transform\
                         (churnData['totaldayminutes']\
                          .values.reshape(-1,1))
churnData['tdcScaled'] = minmaxScaler.fit_transform\
                         (churnData['totaldaycalls']\
                          .values.reshape(-1,1))
churnData['tdchScaled'] = minmaxScaler.fit_transform\
                          (churnData['totaldaycharge']\
                           .values.reshape(-1,1))
churnData['temScaled'] = minmaxScaler.fit_transform\
                         (churnData['totaleveminutes']\
                          .values.reshape(-1,1))
churnData['tecScaled'] = minmaxScaler.fit_transform\
                         (churnData['totalevecalls']\
                          .values.reshape(-1,1))
churnData['techScaled'] = minmaxScaler.fit_transform\
                          (churnData['totalevecharge']\
                           .values.reshape(-1,1))
churnData['tnmScaled'] = minmaxScaler.fit_transform\
                         (churnData['totalnightminutes']\
                          .values.reshape(-1,1))
churnData['tncScaled'] = minmaxScaler.fit_transform\
                         (churnData['totalnightcalls']\
                          .values.reshape(-1,1))
churnData['tnchScaled'] = minmaxScaler.fit_transform\
                          (churnData['totalnightcharge']\
                           .values.reshape(-1,1))
churnData['timScaled'] = minmaxScaler.fit_transform\
                         (churnData['totalintlminutes']\
                          .values.reshape(-1,1))
churnData['ticScaled'] = minmaxScaler.fit_transform\
                         (churnData['totalintlcalls']\
                          .values.reshape(-1,1))
churnData['tichScaled'] = minmaxScaler.fit_transform\
```

```
                                (churnData['totalintlcharge']\
                                 .values.reshape(-1,1))
churnData['ncscScaled'] = minmaxScaler.fit_transform\
                                (churnData\
                                 ['numbercustomerservicecalls']\
                                 .values.reshape(-1,1))
```

4. Since we have saved the transformed numerical features as separate variables, we can drop the original features:

```
# Dropping the original columns
churnData.drop(['accountlength','numbervmailmessages',\
                'totaldayminutes','totaldaycalls',\
                'totaldaycharge','totaleveminutes',\
                'totalevecalls','totalevecharge',\
                'totalnightminutes','totalnightcalls',\
                'totalnightcharge','totalintlminutes',\
                'totalintlcalls','totalintlcharge',\
                'numbercustomerservicecalls'],\
               axis=1, inplace=True)

# Print the head of the data
churnData.head()
```

You should get the following output:

	churn	internationalplan	voicemailplan	alScaled	nvmmScaled	tdmScaled	tdcScaled	tdchScaled	temScaled	tecScaled	techScaled	tnmScaled	tncScaled	tnchScaled
0	No	no	yes	0.524793	0.480769	0.754196	0.666667	0.754183	0.542755	0.582353	0.542866	0.619494	0.520000	0.619584
1	No	no	yes	0.438017	0.500000	0.459744	0.745455	0.459672	0.537531	0.605882	0.537690	0.644051	0.588571	0.644344
2	No	no	no	0.561983	0.000000	0.692461	0.690909	0.692436	0.333242	0.647059	0.333225	0.411646	0.594286	0.411930
3	No	yes	no	0.342975	0.000000	0.851778	0.430303	0.851740	0.170195	0.517647	0.170171	0.498481	0.508571	0.498593
4	No	yes	no	0.305785	0.000000	0.474253	0.684848	0.474230	0.407754	0.717647	0.407959	0.473165	0.691429	0.473270

Figure 13.23: After dropping the original numerical values

From the output, you can see that all the numerical values have been scaled to be in the same range.

5. Next, we transform the categorical data to dummy data using the **`pd.get_dummies()`** function:

```
"""
Converting all the categorical variables to
dummy variables
"""
```

```
churnCat = pd.get_dummies(churnData[['internationalplan',\
                                     'voicemailplan']])
```

6. In this step, we separate the transformed numerical data from the original dataset to later concatenate with the dummy categorical variables:

```
# Separating the numerical data
churnNum = churnData[['alScaled','nvmmScaled',\
                      'tdmScaled','tdcScaled',\
                      'tdchScaled','temScaled',\
                      'tecScaled','techScaled',\
                      'tnmScaled','tncScaled',\
                      'tnchScaled','timScaled',\
                      'ticScaled','tichScaled','ncscScaled']]

churnNum.shape
```

You should get the following output:

```
(5000, 15)
```

7. In this step, we concatenate the transformed categorical variables and numerical variables using the **`pd.concat()`** function to form the **`X`** variable. The target variable **`label`** is stored as the **`Y`** variable:

```
# Merging with the original data frame
# Preparing the X variables
X = pd.concat([churnCat, churnNum], axis=1)
print(X.shape)
# Preparing the Y variable
Y = churnData['churn']
print(Y.shape)
X.head()
```

8. You should get the following output:

```
(5000, 19)
(5000,)
```

	internationalplan_no	internationalplan_yes	voicemailplan_no	voicemailplan_yes	alScaled	nvmmScaled	tdmScaled	tdcScaled	tdchScaled	temScaled	tecScaled
0	1	0	0	1	0.524793	0.480769	0.754196	0.666667	0.754183	0.542755	0.582353
1	1	0	0	1	0.438017	0.500000	0.459744	0.745455	0.459672	0.537531	0.605882
2	1	0	1	0	0.561983	0.000000	0.692461	0.690909	0.692436	0.333242	0.647059
3	0	1	1	0	0.342975	0.000000	0.851778	0.430303	0.851740	0.170195	0.517647
4	0	1	1	0	0.305785	0.000000	0.474253	0.684848	0.474230	0.407754	0.717647

Figure 13.24: Concatenating the transformed categorical variables and numerical variables

9. Next, split the dataset into train and test sets as we have done before:

```
from sklearn.model_selection import train_test_split

# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split\
                                   (X, Y, test_size=0.3, \
                                    random_state=123)
```

10. Now, join the **X** and **y** variables for the training set before resampling:

```
"""
Let us first join the train_x and train_y for
ease of operation
"""
trainData = pd.concat([X_train,y_train],axis=1)

trainData.head()
```

You should get the following output:

	internationalplan_no	internationalplan_yes	voicemailplan_no	voicemailplan_yes	alScaled	nvmmScaled	tdmScaled	tdcScaled	tdchScaled	temScaled	tecScaled
4036	1	0	0	1	0.256198	0.500000	0.609388	0.484848	0.609270	0.695628	0.894118
2883	1	0	1	0	0.504132	0.000000	0.595733	0.296970	0.595716	0.652736	0.688235
4162	0	1	1	0	0.012397	0.000000	0.482788	0.581818	0.482764	0.362387	0.552941
4640	1	0	1	0	0.450413	0.000000	0.714936	0.551515	0.714859	0.569700	0.558824
2430	1	0	0	1	0.491736	0.769231	0.364438	0.600000	0.364458	0.681056	0.458824

Figure 13.25: Joining the X and Y variables before resampling

In this step, we concatenated the **X_train** and **y_train** datasets into one single dataset. This is done to make the resampling process in the subsequent steps easier. To concatenate the two datasets, we use the **.concat()** function from pandas. In the code, we use **axis = 1** to indicate that the concatenation is done horizontally, which is along the columns.

What we will do next is separate the minority class and the majority class. This is required because we have to sample separately from the majority class to make a balanced dataset. To separate the minority class, we have to identify the indexes of the dataset where the dataset has 'yes'. The indexes are identified using the **.index()** function. Once those indexes are identified, they are separated from the main dataset using the **.loc()** function and stored in a new variable for the minority class. The shape of the minority dataset is also printed. A similar process is followed for the majority class and, after these two steps, we have two datasets: one for the minority class, and one for the majority class.

11. Next, find the indexes of the sample dataset where the propensity is **`yes`**:

```
"""
Finding the indexes of the sample data set where
the churn is 'yes'
"""
ind = trainData[trainData['churn']=='Yes'].index
print(len(ind))
```

You should get the following output:

```
490
```

12. Separate the minority classes with the following code snippet:

```
minData = trainData.loc[ind]
print(minData.shape)
```

You should get the following output:

```
(490, 20)
```

13. Now, find the indexes of the majority class:

```
ind1 = trainData[trainData['churn']=='No'].index
print(len(ind1))
```

You should get the following output:

```
3010
```

14. Separate the majority class using the following code snippet:

```
majData = trainData.loc[ind1]
print(majData.shape)
majData.head()
```

You should get the following output:

```
(3010, 20)
```

	internationalplan_no	internationalplan_yes	voicemailplan_no	voicemailplan_yes	alScaled	nvmmScaled	tdmScaled	tdcScaled	tdchScaled	temScaled
4036	1	0	0	1	0.256198	0.500000	0.609388	0.484848	0.609270	0.695628
2883	1	0	1	0	0.504132	0.000000	0.595733	0.296970	0.595716	0.652736
2430	1	0	0	1	0.491736	0.769231	0.364438	0.600000	0.364458	0.681056
449	1	0	0	1	0.322314	0.403846	0.751920	0.478788	0.751841	0.557602
4179	1	0	1	0	0.578512	0.000000	0.613940	0.478788	0.613956	0.309871

Figure 13.26: Separating the majority class

Once the majority class is separated, we can proceed with sampling from the majority class. Once the sampling is done, the shape of the majority class dataset and its head are printed.

Take a random sample equal to the length of the minority class to make the dataset balanced.

15. Extract the sample using the **`.sample()`** function:

```
majSample = majData.sample(n=len(ind),\
                           random_state = 123)
print(majSample.shape)
majSample.head()
```

You should get the following output:

```
(490, 20)
```

	internationalplan_no	internationalplan_yes	voicemailplan_no	voicemailplan_yes	alScaled	nvmmScaled	tdmScaled	tdcScaled	tdchScaled	temScaled	tecScaled
1807	1	0	1	0	0.450413	0.000000	0.557895	0.624242	0.557898	0.549079	0.723529
4578	1	0	1	0	0.475207	0.000000	0.244097	0.533333	0.244143	0.318394	0.658824
355	1	0	1	0	0.123967	0.000000	0.472546	0.636364	0.472557	0.218037	0.547059
23	1	0	1	0	0.454545	0.000000	0.314083	0.624242	0.314090	0.377509	0.600000
1541	1	0	0	1	0.194215	0.692308	0.656899	0.557576	0.656794	0.460819	0.711765

Figure 13.27: Extracting the sample from the data

The number of examples that are sampled is equal to the number of examples in the minority class. This is implemented with the parameters **`(n=len(ind))`**.

Now, we move on to preparing the new training data.

16. Following preparation of the individual dataset, we can now concatenate these individual datasets together using the **`pd.concat()`** function:

```
"""
Concatenating both data sets and then shuffling
the data set
"""
balData = pd.concat([minData,majSample],axis = 0)

print('balanced data set shape',balData.shape)
```

> **NOTE**
>
> In this case, we are concatenating in the vertical direction and, therefore, **`axis = 0`** is used.

You should get the following output:

```
balanced data set shape (980, 20)
```

17. Now, shuffle the dataset so that both the minority and majority classes the using the **`shuffle()`** function:

```
# Shuffling the data set
from sklearn.utils import shuffle
balData = shuffle(balData)
balData.head()
```

The output is as follows (truncated):

	internationalplan_no	internationalplan_yes	voicemailplan_no	voicemailplan_yes	alScaled	nvmmScaled	tdmScaled	tdcScaled	tdchScaled	temScaled
1903	0	1	0	1	0.380165	0.730769	0.642105	0.709091	0.642068	0.328842
4380	1	0	1	0	0.533058	0.000000	0.409104	0.460606	0.409137	0.755568
1209	0	1	0	1	0.590909	0.673077	0.497297	0.769697	0.497323	0.603794
580	1	0	1	0	0.462810	0.000000	0.595733	0.915152	0.595716	0.954908
4119	1	0	0	1	0.557851	0.442308	0.670270	0.393939	0.670181	0.488589

Figure 13.28: Output after shuffling dataset

Now, separate the shuffled dataset into independent variables, **`X_trainNew`**, and dependent variables, **`y_trainNew`**. Separation is to be effected using the index features **0** to **51** for dependent variables using the **`.iloc()`** function in **`pandas`**. The dependent variables are separated by subsetting with the column name **`'y'`**:

```
# Making the new X_train and y_train

X_trainNew = balData.iloc[:,0:19]
X_trainNew.shape
```

You should get the following output:

```
(980, 19)
```

18. Now, make the new **`y_train`**:

```
# Making the new y_train
y_trainNew = balData['churn']
y_trainNew.shape
```

You should get the following output:

```
(980,)
```

Now, fit the model on the new data and generate the confusion matrix and classification report for our analysis.

19. First, define the **`LogisticRegression`** function:

```
from sklearn.linear_model import LogisticRegression

"""
Defining the LogisticRegression function
for Undersampling
"""
churnModel1 = LogisticRegression()
churnModel1.fit(X_trainNew, y_trainNew)
```

20. Importing the library functions and preparing the oversampled dataset for SMOTE instantiation of the library function is done using the command **`sv.SMOTE()`**:

```
import smote_variants as sv
import numpy as np
# Instantiating the SMOTE class
oversampler= sv.SMOTE()
```

21. We will now create new datasets using SMOTE.

    Once the SMOTE method is instantiated, a new training set can be created using the **`.sample()`** function:

> **NOTE**
>
> The training set has to be converted to **`np`** arrays for this to work.

```
# Creating a new training set
X_train_smote, y_train_smote = \
oversampler.sample(np.array(X_train), np.array(y_train))
```

22. The logistic regression model can be fitted as shown here:

```
# Training the model with Logistic regression model
# Defining the LogisticRegression function

churnModel2 = LogisticRegression()
churnModel2.fit(X_train_smote, y_train_smote)
```

23. We now import the library functions and prepare the oversampled dataset for MSMOTE.

    The library function is instantiated using the command, **`sv.MSMOTE()`**:

```
import smote_variants as sv
import numpy as np

# Instantiating the MSMOTE class
oversampler= sv.MSMOTE()
```

24. Once the **`MSMOTE`** method is instantiated, a new training set is created using the **`.sample()`** function. Please note that the training set has to be converted to **`np`** arrays for this to work:

```
# Creating new training sets
X_train_msmote, y_train_msmote = oversampler.sample\
                                 (np.array(X_train), \
                                  np.array(y_train))
```

25. The logistic regression model can be fitted as shown here:

```
# Training the model with Logistic regression model
# Defining the LogisticRegression function
churnModel3 = LogisticRegression()
churnModel3.fit(X_train_msmote, y_train_msmote)
```

26. Now, you need to predict using all three models separately:

```
# Predicting using Under sampler
pred_us = churnModel1.predict(X_test)

# Predicting using SMOTE
pred_smote = churnModel2.predict(X_test)

# Predicting using MSMOTE
pred_msmote = churnModel3.predict(X_test)
```

27. Print the accuracy measures of each model, as shown in the following code snippet:

```
# Printing accuracy of each model
print('Accuracy of Logistic regression model '\
      'prediction on test set for Random Undersampled '\
      'data set: {:.2f}'\
```

```
        .format(churnModel1.score(X_test, y_test)))

print('Accuracy of Logistic regression model '\
      'prediction on test set for SMOTE data set: '\
      '{:.2f}'\
      .format(churnModel2.score(X_test, y_test)))

print('Accuracy of Logistic regression model '\
      'prediction on test set for MSMOTE data set: '\
      '{:.2f}'\
      .format(churnModel3.score(X_test, y_test)))
```

You can expect an output similar to the following:

```
Accuracy of Logistic regression model prediction on test set for Random Undersampled data set: 0.79
Accuracy of Logistic regression model prediction on test set for SMOTE data set: 0.78
Accuracy of Logistic regression model prediction on test set for MSMOTE data set: 0.80
```

Figure 13.29: Accuracy of the models

28. Get the confusion matrix and classification report for each model:

```
"""
Confusion Matrix & Classification reports for the model
"""
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

29. Get the metrics for the undersampling on the dataset:

```
# Metrics for Random undersample data set
print(confusion_matrix(y_test, pred_us))
print(classification_report(y_test, pred_us))
```

You should get an output similar to the following:

```
[[1027  256]
 [  59  158]]
              precision    recall  f1-score   support

          No       0.95      0.80      0.87      1283
         Yes       0.38      0.73      0.50       217

    accuracy                           0.79      1500
   macro avg       0.66      0.76      0.68      1500
weighted avg       0.86      0.79      0.81      1500
```

Figure 13.30: The undersampling output

30. Get the metrics for **SMOTE** on the dataset:

```
# Metrics for SMOTE data set
print(confusion_matrix(y_test, pred_smote))
print(classification_report(y_test, pred_smote))
```

The output is as follows:

```
[[1002  281]
 [  52  165]]
              precision    recall  f1-score   support

          No       0.95      0.78      0.86      1283
         Yes       0.37      0.76      0.50       217

    accuracy                           0.78      1500
   macro avg       0.66      0.77      0.68      1500
weighted avg       0.87      0.78      0.81      1500
```

Figure 13.31: The SMOTE output

31. Get the metrics for **MSMOTE** on the dataset:

```
# Metrics for MSMOTE data set
print(confusion_matrix(y_test, pred_msmote))
print(classification_report(y_test, pred_msmote))
```

The output is as follows:

```
[[1036  247]
 [  55  162]]
              precision    recall  f1-score   support

          No       0.95      0.81      0.87      1283
         Yes       0.40      0.75      0.52       217

    accuracy                           0.80      1500
   macro avg       0.67      0.78      0.70      1500
weighted avg       0.87      0.80      0.82      1500
```

Figure 13.32: The MSMOTE output

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3iRdElQ.
>
> You can also run this example online at https://packt.live/3iVpKui.

In this activity, we have performed data balancing using random undersampling with **SMOTE** and **MSMOTE** for the telecom churn dataset. From the classification report, we can see that **MSMOTE** has the best accuracy, **80%**, compared to **SMOTE** and undersampling techniques, which achieve **79%** and **78%**, respectively. However, we know that it is important to look at the recall values, especially of the minority class. From the recall values, we see that **SMOTE** has the largest value of **76%**. This means that **76%** of customers who are likely to churn have been correctly identified by the model.

Random undersampling and **MSMOTE** have lower recall values of **73%** and **75%**, respectively. We now have a situation where **MSMOTE** has the highest accuracy but a slightly lower recall value and **SMOTE** has the lowest accuracy measure but the highest recall value. In such a situation, we have to look at the f1 scores, which is a weighted score between precision and recall. From all the f1 scores, we see that **MSMOTE** has the highest f1 score of **52%**, with **SMOTE** and random undersampling scoring **50%** each.

Therefore, we can select **MSMOTE** as the best technique for balancing for this context.

CHAPTER 14: DIMENSIONALITY REDUCTION

ACTIVITY 14.01: FITTING A LOGISTIC REGRESSION MODEL ON A HIGH-DIMENSIONAL DATASET

SOLUTION

1. Open a new Colab notebook file. Now, **`import pandas`** to your Colab notebook:

```
import pandas as pd
```

2. Next, set the path of the drive where the **`ad.Data`** file is uploaded, as shown in the following code snippet:

```
# Defining file name of the Github repository
filename = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter14/Dataset/ad.data'
```

3. Read the file using the **`pd.read_csv()`** function from the pandas data frame:

```
adData = pd.read_csv(filename,sep=",",header = None,\
                     error_bad_lines=False)
adData.head()
```

The **`pd.read_csv()`** function's arguments are the filename as a string and the limit separator of a CSV file, which is **`","`**. Please note that as there are no headers for the dataset, we specifically mention it using the **`header = None`** command. The last argument, **`error_bad_lines=False`**, is to skip any errors in the format of the file and then load the data.

After reading the file, the data frame is printed using the **`.head()`** function.

You should get the following output:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	...	1519	1520
0	125	125	1.0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0
1	57	468	8.2105	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0
2	33	230	6.9696	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0
3	60	468	7.8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0
4	60	468	7.8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0

5 rows × 1559 columns

Figure 14.38: Loading data into the Colab notebook

4. Separate the dependent and independent variables from our dataset as shown in the following code snippet:

```
# Separate the dependent and independent variables
# Preparing the X variables
X = adData.loc[:,0:1557]
print(X.shape)
# Preparing the Y variable
Y = adData[1558]
print(Y.shape)
```

You should get the following output:

```
(3279, 1558)
(3279, )
```

As seen earlier, there are **`1559`** features in the dataset. The first **`1558`** features are independent variables. These are separated from the initial **`adData`** data frame using the **`.loc()`** function and giving the indexes of the corresponding features (0 to 1557). The independent variables are loaded into a new variable called **`X`**. The dependent variable that is the label of the dataset is loaded in the **`Y`** variable. The shapes of the dependent and independent variables are also printed.

5. Replace special characters with **`NaN`** values for the first three columns

Replace the special characters in the first three columns, which are of the object type with **`NaN`** values. Replacing special characters with **`NaN`** values makes it easy for further imputation of data.

This is achieved through the following code snippet:

```
"""
Replacing special characters in first 3 columns
which are of type object
"""
for i in range(0,3):
    X[i] = X[i].str.replace("?", 'NaN').values.astype(float)

print(X.head(15))
```

To replace the first three columns, we loop through the columns using the **`for()`** loop and also using the **`range()`** function. Since the first three columns are of the **`object`** or **`string`** type, we use the **`.str.replace()`** function, which stands for "string replace". After replacing the special characters, **`?`**, of the data with **`nan`**, we convert the data to the **`float`** type with the **`.values.astype(float)`** function, which is required for further processing. By printing the first 15 examples, we can see that all special characters have been replaced with **`NaN`** values.

You should get the following output:

	0	1	2	3	4	5	...	1552	1553	1554	1555	1556	1557
0	125.0	125.0	1.0000	1	0	0	...	0	0	0	0	0	0
1	57.0	468.0	8.2105	1	0	0	...	0	0	0	0	0	0
2	33.0	230.0	6.9696	1	0	0	...	0	0	0	0	0	0
3	60.0	468.0	7.8000	1	0	0	...	0	0	0	0	0	0
4	60.0	468.0	7.8000	1	0	0	...	0	0	0	0	0	0
5	60.0	468.0	7.8000	1	0	0	...	0	0	0	0	0	0
6	59.0	460.0	7.7966	1	0	0	...	0	0	0	0	0	0
7	60.0	234.0	3.9000	1	0	0	...	0	0	0	0	0	0
8	60.0	468.0	7.8000	1	0	0	...	0	0	0	0	0	0
9	60.0	468.0	7.8000	1	0	0	...	0	0	0	0	0	0
10	NaN	NaN	NaN	1	0	0	...	0	0	0	0	0	0
11	90.0	52.0	0.5777	1	0	0	...	0	0	0	0	0	0
12	90.0	60.0	0.6666	1	0	0	...	0	0	0	0	0	0
13	90.0	60.0	0.6666	1	0	0	...	0	0	0	0	0	0
14	33.0	230.0	6.9696	1	0	0	...	0	0	0	0	0	0

Figure 14.39: After replacing special characters with NaN

6. Now, replace special characters with integer features.

   Like *Step 5*, let's also replace the special characters from the features of the **`int64`** data type with the following code snippet:

```
"""
Replacing special characters in the remaining
columns which are of type integer
"""
for i in range(3,1557):
    X[i] = X[i].replace("?", 'NaN').values.astype(float)
```

> **NOTE**
>
> For the integer features, we do not have **`.str`** before the **`.replace ()`** function, as these features are integer values and not string values.

7. Impute the mean of each column for **NaN** values.

   Now that we have replaced special characters in the data with **NaN** values we can use the **fillna()** function in pandas to replace the **NaN** values with the mean of the column. This is executed using the following code snippet:

```
import numpy as np
# Impute the 'NaN' with mean of the values
for i in range(0,1557):
    X[i] = X[i].fillna(X[i].mean())
print(X.head(15))
```

   From the preceding code snippet, the **.mean()** function calculates the mean of each column and then replaces the **NaN** values with the mean of the column.

   You should get the following output:

```
           0           1         2     3     4     ...  1553  1554  1555  1556  1557
0   125.000000  125.000000  1.000000   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
1    57.000000  468.000000  8.210500   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
2    33.000000  230.000000  6.969600   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
3    60.000000  468.000000  7.800000   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
4    60.000000  468.000000  7.800000   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
5    60.000000  468.000000  7.800000   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
6    59.000000  460.000000  7.796600   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
7    60.000000  234.000000  3.900000   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
8    60.000000  468.000000  7.800000   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
9    60.000000  468.000000  7.800000   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
10   64.021886  155.344828  3.911953   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
11   90.000000   52.000000  0.577700   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
12   90.000000   60.000000  0.666600   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
13   90.000000   60.000000  0.666600   1.0   0.0  ...   0.0   0.0   0.0   0.0     0
14   33.000000  230.000000  6.969600   1.0   0.0  ...   0.0   0.0   0.0   0.0     0

[15 rows x 1558 columns]
```

Figure 14.40: Mean of the columns of NaN

8. Scale the dataset using the **minmaxScaler()** function.

   As we learned in *Chapter 3*, *Binary Classification*, the scaling of data would help in the modeling step. Let's scale the dataset using the **minmaxScaler()** function, as we learned in *Chapter 3*, *Binary Classification*.

This is described in the following code snippet:

```
# Scaling the data sets
# Import library function
from sklearn import preprocessing

# Creating the scaling function
minmaxScaler = preprocessing.MinMaxScaler()

# Transforming with the scaler function
X_tran = pd.DataFrame(minmaxScaler.fit_transform(X))

# Printing the output
X_tran.head()
```

You should get the following output. Here, we have displayed the first 24 columns:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0.194053	0.194053	0.016642	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.087637	0.730829	0.136820	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.050078	0.358372	0.116138	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.092332	0.730829	0.129978	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.092332	0.730829	0.129978	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 1558 columns

Figure 14.41: Scaling the dataset using the minmaxScaler() function

9. Replicate the columns of the database **300** times using the **pd.np.tile()** function:

```
# Creating a high dimension data set
X_hd = pd.DataFrame(pd.np.tile(X_tran, (1, 300)))

# Printing the dimension of the data set
X_hd.shape
```

In the preceding code snippet, you have used the **pd.np.tile()** function to scale the **X_train** dataset 300 times.

You should get the following output:

```
(3279, 467400)
```

From the output, we can see that **467400** is the number of features in this dataset after scaling by a factor of **300** and **3279** is the number of rows.

10. Split the dataset into training and testing sets:

```
from sklearn.model_selection import train_test_split

# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split\
                                   (X_hd, Y, test_size=0.3, \
                                    random_state=123)
```

In the preceding code snippet, you have used the **train_test_split()** function to split the dataset into training and testing sets.

> **NOTE**
>
> If you are unable to run the previous step due to insufficient RAM, you can reduce the parameter in **X_hd = pd.DataFrame(pd.np.tile(X_tran, (1, 300)))** from **300** to a lower value (e.g. **100**).

11. Fit a logistic regression model and note the time taken for the model building, as shown in the following code snippet:

```
from sklearn.linear_model import LogisticRegression
import time

# Defining the LogisticRegression function
benchmarkModel = LogisticRegression()

# Starting a timing function
t0=time.time()
# Fitting the model
benchmarkModel.fit(X_train, y_train)
# Finding the end time
print("Total training time:", \
      round(time.time()-t0, 3), "s")
```

In the preceding code snippet, you have used the **time()** function to note the time of the model fitting. The model fitting is done using a logistic regression function using the **.fit()** function on the training set.

You should get a similar output:

```
Total training time: 23.86 s
```

As seen from the output, the new dataset has taken **23.86** seconds to fit the model which can be attributed to the large dimension of the dataset. Note the use of the RAM from the indicator.

In the top-right-hand corner of your Colab notebook is the runtime resource indicator. Notice the change in the color of your RAM indicator.

You should see a similar indication on your RAM, which indicates a high utilization of resources.:

Figure 14.42: RAM utilization of Colab

12. Now predict on the test set and print the accuracy measures:

```
# Predicting using the model
pred = benchmarkModel.predict(X_test)
print('Accuracy of Logistic regression model '\
      'prediction on test set: {:.2f}'\
      .format(benchmarkModel.score(X_test, y_test)))
```

In the preceding code snippet, you have used the **.predict()** function to generate the predictions on the test set. The accuracy scores are then printed.

You should get output similar to this:

```
Accuracy of Logistic regression model prediction on test set: 0.97
```

13. Now, print the confusion matrix and classification report:

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
# Confusion Matrix for the model
print(confusion_matrix(y_test, pred))
# Classification report for the model
print(classification_report(y_test, pred))
```

In the preceding code snippet, you have printed the confusion matrix using the **`confusion_matrix()`** function.

You should get the following output:

```
[[112  14]
 [ 18 840]]
              precision    recall  f1-score   support

          ad.       0.86      0.89      0.88       126
       nonad.       0.98      0.98      0.98       858

    accuracy                           0.97       984
   macro avg       0.92      0.93      0.93       984
weighted avg       0.97      0.97      0.97       984
```

Figure 14.43: Confusion matrix and the classification report results

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3aFdXxa.
>
> This section does not currently have an online interactive example, but can be run as usual on Google Colab.

ACTIVITY 14.02: COMPARISON OF DIMENSIONALITY REDUCTION TECHNIQUES ON THE ENHANCED ADS DATASET

SOLUTION

1. Open a new Colab notebook file. Now, **`import pandas`** into your Colab notebook:

```
import pandas as pd
```

2. Next, set the path of the drive where the **`ad.Data`** file is uploaded:

```
# Defining file name of the github repository
filename = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter14/Dataset/ad.data'
```

3. Read the file using the **`pd.read_csv()`** function from the pandas data frame:

```
adData = pd.read_csv(filename,sep=",",header = None,\
                     error_bad_lines=False)
adData.head()
```

The **`pd.read_csv()`** function's arguments are the filename as a string and the limit separator of a CSV file, which is **`","`**. Please note that as there are no headers for the dataset, we specify this using the **`header = None`** command. The last argument, **`error_bad_lines=False`**, is to skip any errors in the format of the file and then load the data.

After reading the file, the data frame is printed using the **`.head()`** function.

You should get the following output:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	...	1519	1520
0	125	125	1.0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0
1	57	468	8.2105	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0
2	33	230	6.9696	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0
3	60	468	7.8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0
4	60	468	7.8	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	...	0	0

5 rows × 1559 columns

Figure 14.44: Loading data into the Colab notebook

4. Now separate the dependent and independent variables:

```
# Separate the dependent and independent variables
# Preparing the X variables
X = adData.loc[:,0:1557]
print(X.shape)
# Preparing the Y variable
Y = adData[1558]
print(Y.shape)
```

You should get the following output

```
(3279, 1558)
(3279,)
```

As seen earlier, there are **`1559`** features in the dataset. The first **`1558`** features are the independent variables. These are separated from the initial **`adData`** data frame using the **`.loc()`** function by giving the indexes of the corresponding columns (**`0`** to **`1557`**). The independent variables are loaded into a new variable called **`X`**. The dependent variable, which is the label of the dataset, is loaded in the **`Y`** variable. The shapes of the dependent and independent variables are also printed.

5. Let's replace the special characters in the first three columns that are of the object type with **NaN** values. Replacing special characters with **NaN** values makes it easy to further impute data.

   This is achieved in the first part of the following code snippet:

```
import numpy as np
"""
Replacing special characters in first 3
columns which are of type object
"""
for i in range(0,3):
    X[i] = X[i].str.replace("?", 'NaN').values.astype(float)
```

   To replace the first three columns, we loop through the columns using a **for()** loop and also using the **range()** function. Since the first three columns are of the object or string type, we use the **.str.replace()** function, which stands for "string replace". After replacing the special characters, **?**, of the data with **nan**, we convert the data to the float type with the **.values.astype(float)** function, which is required for further processing.

   Similar to the previous step, the next code snippet is to replace the special characters from the features of data type **int64**:

```
"""
Replacing special characters in the remaining
columns which are of type integer
"""
for i in range(3,1557):
    X[i] = X[i].replace("?", 'NaN').values.astype(float)
# Imputing the 'nan' with mean of the values
for i in range(0,1557):
    X[i] = X[i].fillna(X[i].mean())
```

> **NOTE**
>
> For the integer features, we do not have **.str** before the **.replace()** function because these features are integer values and not string values.

Now that we have replaced special characters in the data with **NaN** values, we can use **fillna()** function in pandas to replace the **NaN** values with the mean of the column. This is executed in the third part of the code snippet. The **.mean()** function calculates the mean of each column and then replaces the **nan** values with the mean of the column.

6. Scale the dataset using the **MinMaxScaler()** function.

   As we learned in *Chapter 3*, *Binary Classification,* scaling data helps in the modeling step. Let's scale the dataset using the **MinMaxScaler()** function.

   This is described in the following code snippet:

```
# Scaling the data sets
# Import library function
from sklearn import preprocessing

# Creating the scaling function
minmaxScaler = preprocessing.MinMaxScaler()

# Transforming with the scaler function
X_tran = pd.DataFrame(minmaxScaler.fit_transform(X))

# Printing the output
X_tran.head()
```

   You should get the following output:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0.194053	0.194053	0.016642	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.087637	0.730829	0.136820	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.050078	0.358372	0.116138	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.092332	0.730829	0.129978	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.092332	0.730829	0.129978	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 14.45: Output with the first 24 columns

7. Create a high-dimensional dataset using the **pd.np.tile()** function and a factor of **2**:

```
# Creating a high dimensional dataset
X_hd = pd.DataFrame(pd.np.tile(X_tran, (1, 2)))
```

```
# Printing the dimension of the data set
X_hd.shape
```

In this step, we have used the **`np.tile()`** function to scale the dimension of the dataset by a factor of **`2`**. In the code snippet, **`(1,2)`** means that the columns are replicated twice.

You should get the following output:

```
(3279, 3116)
```

8. Define the mean and standard deviation for sampling:

```
# Defining the mean and standard deviation
mu, sigma = 0, 0.1
```

In this step, we assume the mean and standard deviation of the distribution from which we are going to sample random data. The sampling is done in the next step.

9. Generate samples from the normal distribution:

```
# Generating samples from the distribution
noise = np.random.normal(mu, sigma, [3279,3116])
noise.shape
```

You should get the following output:

```
(3279, 3116)
```

What we will do in this step is to sample some data points with the same shape as our data frame. Let's sample some data points from a normal distribution that has mean **`0`** and standard deviation **`0.1`**. We covered the normal distribution in *Chapter 3, Binary Classification*. In that chapter, you may remember that we said that a normal distribution has got two parameters. The first one is the mean, which is the average of all the data in the distribution, and the second one is the standard deviation, which is a measure of how spread out the data points are. We defined the mean and standard deviation in *Step 8*. By setting a mean and standard deviation, we will be able to draw samples from a normal distribution using the **`np.random.normal()`** function. The arguments that we have to give for this function are the mean, the standard deviation, and the shape of the new dataset.

10. Create the new dataset by adding the high-dimensional dataset to the random samples:

```
# Creating a new data set by adding noise
X_new = X_hd + noise
```

11. The dataset is then split into train and test sets using the **`train_test_split()`** function:

```
from sklearn.model_selection import train_test_split
# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split\
                                   (X_new, Y, test_size=0.3,\
                                    random_state=123)
print('Training set shape',X_train.shape)
print('Test set shape',X_test.shape)
```

You should get the following output:

```
Training set shape (2295, 3116)
Test set shape (984, 3116)
```

12. The backward elimination method works by providing two arguments to the **`RFE()`** function, which are the model we want to try (logistic regression in our case) and the number of features we want the dataset to be reduced to. This is implemented as follows:

```
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import RFE

# Defining the Classification function
backModel = LogisticRegression()
"""
Reducing dimensionality to 300 features for
backward elimination model
"""
rfe = RFE(backModel, 300)
```

In this implementation, the number of features that we have given, **`300`**, is identified through trial and error. The process is to first assume an arbitrary number of features and then, based on the final metrics, arrive at the optimum number of features for the model. In this implementation, our first assumption of **`300`** implies that we want the backward elimination model to start eliminating features until we get the best **`300`** features.

13. Now, fit the backward elimination method on the higher dimension dataset. We will also note the time it takes for backward elimination to work. This is implemented using the following code snippet:

```
# Fitting the rfe for selecting the top 300 features
import time
t0 = time.time()
rfe = rfe.fit(X_train, y_train)
t1 = time.time()
print("Backward Elimination time:", \
      round(t1-t0, 3), "s")
```

Fitting the backward elimination method is done using the **`.fit()`** function. We give the independent and dependent training sets. Please note that the backward elimination method is a compute-intensive process, and therefore this process will take a lot of time to execute. The greater the number of features, the more time it will take.

The time taken for the backward elimination process is at the end of the notifications.

You should get a similar output to this:

```
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver will be
  FutureWarning)
Backward Elimination time: 2392.68 s
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver will be
  FutureWarning)
```

Figure 14.46: Execution time for the backward elimination

Earlier, in *Step 12*, we identified the top **`300`** features through backward elimination.

14. Now reduce the train and test sets to those top **`300`** features. This is done using the **`.transform()`** function. This is implemented using the following code snippet:

```
# Transforming both train and test sets
X_train_tran = rfe.transform(X_train)
X_test_tran = rfe.transform(X_test)
print("Training set shape",X_train_tran.shape)
print("Test set shape",X_test_tran.shape)
```

You should get the following output:

```
Training set shape (2295, 300)
Test set shape (984, 300)
```

15. Now, fit a logistic regression model on the training set and note the time:

```
# Fitting the logistic regression model
import time
# Defining the LogisticRegression function
RfeModel = LogisticRegression()
# Starting a timing function
t0=time.time()
# Fitting the model
RfeModel.fit(X_train_tran, y_train)
# Finding the end time

print("Total training time:", \
      round(time.time()-t0, 3), "s")
```

You should get the following output:

```
Total training time: 0.085 s
```

16. Now, predict on the test set and print the accuracy metrics:

```
# Predicting on the test set and getting the accuracy
pred = RfeModel.predict(X_test_tran)

print('Accuracy of Logistic regression model after '\
      'backward elimination: {:.2f}'\
      .format(RfeModel.score(X_test_tran, y_test)))
```

You should get a similar output to this:

```
Accuracy of Logistic regression model after backward elimination:
0.97
```

17. Print the confusion matrix after the predictions:

```
# Printing the Confusion matrix
from sklearn.metrics import confusion_matrix
confusionMatrix = confusion_matrix(y_test, pred)
print(confusionMatrix)
```

You should get a similar output to this:

```
[[ 95  31]
 [  1 857]]
```

Figure 14.47: Expected confusion matrix

18. Print the classification report:

```
from sklearn.metrics import classification_report
# Getting the Classification_report
print(classification_report(y_test, pred))
```

You should get the following output:

```
              precision    recall  f1-score   support

         ad.       0.99      0.75      0.86       126
      nonad.       0.97      1.00      0.98       858

    accuracy                           0.97       984
   macro avg       0.98      0.88      0.92       984
weighted avg       0.97      0.97      0.97       984
```

Figure 14.48: Expected classification matrix

19. Now we shall implement forward selection. The first step is to import the **`SelectKBest`** feature selection function. The argument we give to this function is the number of features we want. The selection of features is arrived at through experimentation and, as a first step, we assume a threshold value. In this example, we assume a threshold value of **`300`**. This is implemented using the following code snippet:

```
from sklearn.feature_selection import SelectKBest

# feature extraction
feats = SelectKBest(k=300)
```

20. Based on the threshold set of features we defined, we have to fit the training set and get the best set of threshold features. Fitting on the training set is done using the **`.fit()`** function. We also note the time it takes to find the best set of features. This is executed using the following code snippet:

```
# Fitting the features for training set
import time
t0 = time.time()
fit = feats.fit(X_train, y_train)
t1 = time.time()
print("Forward selection fitting time:", \
      round(t1-t0, 3), "s")
```

You should get the following output.

```
Forward selection fitting time: 0.098 s
```

21. Next, modify our training and test sets so that they have only those selected features. This is accomplished using the **`.transform()`** function:

```
# Creating new training set and test sets
features_train = fit.transform(X_train)
features_test = fit.transform(X_test)
```

22. Verify the shapes of the train and test sets before transformation and after transformation:

```
"""
Printing the shape of train and test sets
before transformation
"""
print('Train shape before transformation',X_train.shape)
print('Test shape before transformation',X_test.shape)

"""
Printing the shape of train and test sets
after transformation
"""
print('Train shape after transformation',\
      features_train.shape)
print('Test shape after transformation',\
      features_test.shape)
```

You should get the following output:

```
Train shape before transformation (2295, 3116)
Test shape before transformation (984, 3116)
Train shape after transformation (2295, 300)
Test shape after transformation (984, 300)
```

Figure 14.49: Output after verifying the shape of the train and test sets

23. Fit the logistic regression model on the transformed dataset and note the time it takes to fit the model:

```
# Fitting a Logistic Regression Model
from sklearn.linear_model import LogisticRegression
import time

t0 = time.time()

forwardModel = LogisticRegression()
forwardModel.fit(features_train, y_train)

t1 = time.time()
print("Total training time:", round(t1-t0, 3), "s")
```

You should get the following output:

```
Total training time: 0.114 s
```

24. Let's now predict on the test set and print the accuracy metrics:

```
# Predicting with the forward model
pred = forwardModel.predict(features_test)
print('Accuracy of Logistic regression model '\
      'prediction on test set: {:.2f}'\
      .format(forwardModel.score(features_test, \
                                 y_test)))
```

You should get the following output:

```
Accuracy of Logistic regression model prediction on test set: 0.97
```

25. Now, print the confusion matrix:

```
# Generating confusion matrix
from sklearn.metrics import confusion_matrix

confusionMatrix = confusion_matrix(y_test, pred)
print(confusionMatrix)
```

You should get something like the following output:

```
[[ 95  31]
 [  2 856]]
```

Figure 14.50: Expected confusion matrix

26. Now, print the classification report:

```
from sklearn.metrics import classification_report
# Getting the Classification_report
print(classification_report(y_test, pred))
```

You should get the following output:

```
              precision    recall  f1-score   support

          ad.       0.97      0.75      0.85       126
       nonad.       0.97      1.00      0.98       858

    accuracy                           0.97       984
   macro avg       0.97      0.88      0.91       984
weighted avg       0.97      0.97      0.96       984
```

Figure 14.51: Expected classification matrix

27. Now we shall implement PCA. First, we define the number of components we desire using the **`n_components`** argument. After that, fit the PCA function on the training set. This is done using the **`.fit()`** function as shown in the following snippet. We will also note the time it takes to fit the PCA model on the dataset:

```
from sklearn.decomposition import PCA
import time
t0 = time.time()
```

```
pca = PCA(n_components=300)
# Fitting the PCA on the training set
pca.fit(X_train)
t1 = time.time()
print("PCA fitting time:", round(t1-t0, 3), "s")
```

You should get a similar output to this:

```
PCA fitting time: 2.843 s
```

28. Now transform the training and test sets with the **300** principal components:

```
# Transforming training set and test set
X_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)
```

29. Verify the shapes of the train and test sets before transformation and after transformation:

```
print("original shape of Training set:   ", \
      X_train.shape)
print("original shape of Test set:   ", \
      X_test.shape)
print("Transformed shape of training set:", \
      X_pca.shape)
print("Transformed shape of test set:", \
      X_test_pca.shape)
```

You should get a similar output to this:

```
original shape of Training set:    (2295, 3116)
original shape of Test set:    (984, 3116)
Transformed shape of training set: (2295, 300)
Transformed shape of test set: (984, 300)
```

Figure 14.52: Expected shapes of the train and test sets

30. Fit the logistic regression model on the transformed dataset and note the time it takes to fit the model:

```
from sklearn.linear_model import LogisticRegression
import time
pcaModel = LogisticRegression()

t0 = time.time()
pcaModel.fit(X_pca, y_train)
t1 = time.time()
print("Total training time:", round(t1-t0, 3), "s")
```

You should get the following output:

```
Total training time: 0.138 s
```

31. Predict the test set and print the accuracy metrics:

```
# Predicting with the pca model
pred = pcaModel.predict(X_test_pca)
print('Accuracy of Logistic regression model '\
      'prediction on test set: {:.2f}'\
      .format(pcaModel.score(X_test_pca, y_test)))
```

You should get the following output:

```
Accuracy of Logistic regression model prediction on test set: 0.97
```

32. Now print the confusion matrix and the classification report after the prediction:

```
# Generating confusion matrix
from sklearn.metrics import confusion_matrix

confusionMatrix = confusion_matrix(y_test, pred)
print(confusionMatrix)

from sklearn.metrics import classification_report
# Getting the Classification_report
print(classification_report(y_test, pred))
```

You should get something like the following output:

```
[[100  26]
 [  3 855]]
```

	precision	recall	f1-score	support
ad.	0.99	0.78	0.87	126
nonad.	0.97	1.00	0.98	858
accuracy			0.97	984
macro avg	0.98	0.89	0.93	984
weighted avg	0.97	0.97	0.97	984

Figure 14.53: Expected classification matrix

33. Now, let's implement ICA. Let's load the ICA function, **`FastICA`**, and then define the number of components we require:

```
# Defining the ICA with number of components
from sklearn.decomposition import FastICA
ICA = FastICA(n_components=300, random_state=123)
```

Once the ICA method is defined, we will fit the method on the training set and also transform the training set to get a new training set with the required number of components. We will also note the time taken for fitting and transforming:

```
"""
Fitting the ICA method and transforming the
training set and noting the time
"""
import time
t0 = time.time()
X_ica=ICA.fit_transform(X_train)
t1 = time.time()
print("ICA fitting time:", round(t1-t0, 3), "s")
```

In the code, the **`.fit()`** function is used to fit on the training set and the **`transform()`** method is used to get a new training set with the required number of features.

You should get the following output:

```
ICA fitting time: 27.562 s
```

34. Now, transform the test set with the **300** components:

```
# Transforming the test set
X_test_ica=ICA.transform(X_test)
```

35. Let's verify the shapes of the train and test sets before transformation and after transformation:

```
print("original shape of Training set:   ", X_train.shape)
print("original shape of Test set:   ", X_test.shape)
print("Transformed shape of training set:", X_ica.shape)
print("Transformed shape of test set:", X_test_ica.shape)
```

You should get the following output:

```
original shape of Training set:    (2295, 3116)
original shape of Test set:    (984, 3116)
Transformed shape of training set: (2295, 300)
Transformed shape of test set: (984, 300)
```

Figure 14.54: Expected shapes of the train and test sets

36. Fit the logistic regression model on the transformed dataset and note the time it takes to fit the model:

```
from sklearn.linear_model import LogisticRegression
import time
icaModel = LogisticRegression()

t0 = time.time()
icaModel.fit(X_ica, y_train)
t1 = time.time()
print("Total training time:", round(t1-t0, 3), "s")
```

You should get the following output:

```
Total training time: 0.046 s
```

37. Predict on the test set and print the accuracy metrics:

```
# Predicting with the ica model
pred = icaModel.predict(X_test_ica)
print('Accuracy of Logistic regression model '\
      'prediction on test set: {:.2f}'\
      .format(icaModel.score(X_test_ica, y_test)))
```

You should get the following output:

```
Accuracy of Logistic regression model prediction on test set: 0.87
```

38. Print the confusion matrix:

```
# Generating confusion matrix
from sklearn.metrics import confusion_matrix
confusionMatrix = confusion_matrix(y_test, pred)
print(confusionMatrix)
```

You should get the following output:

```
[[  0 126]
 [  0 858]]
```

Figure 14.55: Expected confusion matrix

39. Print the classification report:

```
from sklearn.metrics import classification_report
# Getting the Classification_report
print(classification_report(y_test, pred))
```

You should get the following output:

```
              precision    recall  f1-score   support

         ad.       0.00      0.00      0.00       126
      nonad.       0.87      1.00      0.93       858

    accuracy                           0.87       984
   macro avg       0.44      0.50      0.47       984
weighted avg       0.76      0.87      0.81       984
```

Figure 14.56: Expected classification matrix

40. Finally, let's implement factor analysis. An important step in factor analysis is defining the number of factors in a dataset. In our case, we will arbitrarily assume **30** factors. This is implemented as shown in the following code snippet:

```
# Defining the number of factors
from sklearn.decomposition import FactorAnalysis
fa = FactorAnalysis(n_components = 30,random_state=123)
```

The number of factors are defined through the **n_components** argument. We also define a random state for reproducibility, **123**.

41. Fit the method on the training set and also transform the training set to get a new training set with the required number of factors. We will also note the time it takes to fit the required number of factors:

```
"""
Fitting the Factor analysis method and transforming
the training set
"""
import time
t0 = time.time()
X_fac=fa.fit_transform(X_train)
t1 = time.time()
print("Factor analysis fitting time:", \
      round(t1-t0, 3), "s")
```

In the code, the **.fit()** function is used to fit on the training set and the **transform()** method is used to get a new training set with the required number of factors.

You should get the following output:

```
Factor analysis fitting time: 3.802 s
```

42. We now transform the test set with the same number of factors:

```
# Transforming the test set
X_test_fac=fa.transform(X_test)
```

43. Verify the shapes of the train and test sets before transformation and after transformation:

```
print("original shape of Training set:   ", X_train.shape)
print("original shape of Test set:   ", X_test.shape)
```

```
print("Transformed shape of training set:", X_fac.shape)
print("Transformed shape of test set:", X_test_fac.shape)
```

You should get the following output:

```
original shape of Training set:    (2295, 3116)
original shape of Test set:    (984, 3116)
Transformed shape of training set: (2295, 30)
Transformed shape of test set: (984, 30)
```

Figure 14.57: Expected shapes of the train and test sets

44. Now fit the logistic regression model on the transformed dataset and note the time it takes to fit the model:

```
from sklearn.linear_model import LogisticRegression
import time

facModel = LogisticRegression()

t0 = time.time()
facModel.fit(X_fac, y_train)
t1 = time.time()
print("Total training time:", round(t1-t0, 3), "s")
```

You should get the following output:

```
Total training time: 0.023 s
```

45. Predict on the test set and print the accuracy metrics:

```
# Predicting with the factor analysis model
pred = facModel.predict(X_test_fac)
print('Accuracy of Logistic regression model '\
      'prediction on test set: {:.2f}'\
      .format(facModel.score(X_test_fac, y_test)))
```

You should get the following output:

```
Accuracy of Logistic regression model prediction on test set: 0.96
```

46. Let's now print the confusion matrix:

```
# Generating confusion matrix
from sklearn.metrics import confusion_matrix
```

```
confusionMatrix = confusion_matrix(y_test, pred)
print(confusionMatrix)
```

You should get something like the following output:

```
[[ 86  40]
 [  2 856]]
```

Figure 14.58: Expected confusion matrix

47. Verify the classification report:

```
from sklearn.metrics import classification_report
# Getting the Classification_report
print(classification_report(y_test, pred))
```

You should get the following output:

```
              precision    recall  f1-score   support

         ad.       0.98      0.68      0.80       126
      nonad.       0.96      1.00      0.98       858

    accuracy                           0.96       984
   macro avg       0.97      0.84      0.89       984
weighted avg       0.96      0.96      0.95       984
```

Figure 14.59: Expected classification matrix

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/31d6fHx.
>
> You can also run this example online at https://packt.live/3gdv8XS.

CHAPTER 15: ENSEMBLE LEARNING

ACTIVITY 15.01: FITTING A LOGISTIC REGRESSION MODEL ON CREDIT CARD DATA

SOLUTION

1. Execute all the steps from *Exercise 15.01* to split the dataset into train and test sets.
2. Next, import the necessary libraries and fit a logistic regression model on the dataset, as shown in the following code snippet:

```
from sklearn.linear_model import LogisticRegression
# Defining the LogisticRegression function
benchmarkModel = LogisticRegression()

# Fitting the model
benchmarkModel.fit(X_train, y_train)
```

3. Predict the test set:

```
# Prediction and accuracy metrics
pred = benchmarkModel.predict(X_test)
print('Accuracy of Logistic regression model '\
      'prediction on test set: {:.2f}'\
      .format(benchmarkModel.score(X_test, y_test)))
```

You should get an output similar to the following:

```
Accuracy of Logistic regression model prediction on test set: 0.89
```

4. Finally, derive the classification report and confusion matrix for the model, as shown in the following code snippet:

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

# Confusion Matrix for the model
print(confusion_matrix(y_test, pred))

# Classification report for the model
print(classification_report(y_test, pred))
```

You should get an output similar to the following:

```
[[93 14]
 [ 8 81]]
              precision    recall  f1-score   support

           0       0.92      0.87      0.89       107
           1       0.85      0.91      0.88        89

    accuracy                           0.89       196
   macro avg       0.89      0.89      0.89       196
weighted avg       0.89      0.89      0.89       196
```

Figure 15.36: Expected confusion matrix

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2ElNclg.
>
> You can also run this example online at https://packt.live/3l1bC4q.

ACTIVITY 15.02: COMPARISON OF ADVANCED ENSEMBLE TECHNIQUES

SOLUTION

1. Execute all the steps from *Exercise 15.07* until you have split the dataset into train and test sets.
2. Bagging: Define the base learner, create the meta learner, fit the model, generate predictions, and print the confusion matrix and classification report:

```
# Defining the base learner
from sklearn.linear_model import LogisticRegression
bl1 = LogisticRegression(random_state=123)
# Creating the bagging meta learner
from sklearn.ensemble import BaggingClassifier
baggingLearner = BaggingClassifier(base_estimator=bl1, \
                                   n_estimators=15, \
                                   max_samples=0.7, \
                                   max_features=0.8)
```

```
# Fitting the model using the meta learner
model = baggingLearner.fit(X_train, y_train)
# Predicting on the test set using the model
pred = model.predict(X_test)

# Printing the confusion matrix
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test, pred))

# Printing the classification report
from sklearn.metrics import classification_report
print(classification_report(y_test, pred))
```

3. Boosting: Define the base learner, define the meta learner, fit the model, generate the predictions, and print the confusion matrix and classification report:

```
# Defining the base learner
from sklearn.ensemble import RandomForestClassifier
bl1 = RandomForestClassifier(random_state=123)
# Defining the boosting meta learner
from sklearn.ensemble import AdaBoostClassifier
boosting = AdaBoostClassifier(base_estimator=bl1, \
                              n_estimators=300)
# Fitting the model on the training set
model = boosting.fit(X_train, y_train)
# Getting the predictions from the boosting model
pred = model.predict(X_test)
# Printing the confusion matrix
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test, pred))
# Printing the classification report
from sklearn.metrics import classification_report
print(classification_report(y_test, pred))
```

4. Stacking: Define the base learners and meta learner, create the stacking classifier, fit the model, generate the predictions, and print the confusion matrix and classification report:

```
# Importing the meta learner and base learners
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.ensemble import RandomForestClassifier
bl1 = KNeighborsClassifier(n_neighbors=5)
bl2 = LogisticRegression(random_state=123)
ml = RandomForestClassifier(random_state=123)
# Creating the stacking classifier
from mlxtend.classifier import StackingClassifier
stackclf = StackingClassifier(classifiers=[bl1, bl2], \
                              meta_classifier=ml)
# Fitting the model on the training set
model = stackclf.fit(X_train, y_train)
# Generating predictions on test set
pred = model.predict(X_test)
# Printing the confusion matrix
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test, pred))
# Printing the classification report
from sklearn.metrics import classification_report
print(classification_report(y_test, pred))
```

The output for bagging would be as follows:

```
[[93 14]
 [ 6 83]]

              precision    recall  f1-score   support

           0       0.94      0.87      0.90       107
           1       0.86      0.93      0.89        89

    accuracy                           0.90       196
   macro avg       0.90      0.90      0.90       196
weighted avg       0.90      0.90      0.90       196
```

Figure 15.37: Output for bagging

The output for boosting would be as follows:

```
[[97 10]
 [ 8 81]]
              precision    recall  f1-score   support

           0       0.92      0.91      0.92       107
           1       0.89      0.91      0.90        89

    accuracy                           0.91       196
   macro avg       0.91      0.91      0.91       196
weighted avg       0.91      0.91      0.91       196
```

Figure 15.38: Output for boosting

The output for stacking would be as follows:

```
[[99  8]
 [18 71]]
              precision    recall  f1-score   support

           0       0.85      0.93      0.88       107
           1       0.90      0.80      0.85        89

    accuracy                           0.87       196
   macro avg       0.87      0.86      0.86       196
weighted avg       0.87      0.87      0.87       196
```

Figure 15.39: Output for stacking

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3iYcVzi.
>
> You can also run this example online https://packt.live/3gbMmFe.

CHAPTER 16: MACHINE LEARNING PIPELINES

ACTIVITY 16.01: COMPLETE ML WORKFLOW IN A PIPELINE

SOLUTION:

1. Open a new Colab notebook
2. Load the dataset from the GitHub repository:

> **NOTE**
>
> The dataset to be used in this activity can be found on our GitHub repository at https://packt.live/37DJcpO

```
import pandas as pd
#Loading data from GitHub repository
filename = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter16/Dataset'\
           '/processed.cleveland.data'
```

3. Read the data using **`pandas`** and then impute **`NA`** values where there are missing values or special characters such as **`?`**:

```
# Loading the data using pandas

heartData = pd.read_csv(filename,sep=",",\
                        header = None,na_values = "?")
heartData.head()
```

You should get the following output:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	6.0	0
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	3.0	2
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	7.0	1
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	3.0	0
4	41.0	0.0	2.0	130.0	204.0	0.0	2.0	172.0	0.0	1.4	1.0	0.0	3.0	0

Figure 16.24: Data read using pandas

4. Define the names of the columns using the **`.columns`** function. Assign the names as given in the following list: [**`'age'`**,**`'sex'`**, **`'cp'`**, **`'trestbps'`**,**`'chol'`**,**`'fbs'`**,**`'restecg'`**,**`'thalach'`**,**`'exang'`**, **`'oldpeak'`**,**`'slope'`**,**`'ca'`**,**`'thal'`**,**`'label'`**]:

```
heartData.columns = ['age','sex', 'cp', 'trestbps',\
                     'chol','fbs','restecg','thalach',\
                     'exang','oldpeak','slope','ca',\
                     'thal','label']
heartData.head()
```

You should get the following output:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	label
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	6.0	0
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	3.0	2
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	7.0	1
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	3.0	0
4	41.0	0.0	2.0	130.0	204.0	0.0	2.0	172.0	0.0	1.4	1.0	0.0	3.0	0

Figure 16.25: Column names defined

5. Change the classes of all values other than **0** in the **`label`** column to **1**, similar to what was done in the credit card dataset. This is done to make this problem a binary classification problem with two labels:

```
# Changing the Classes to 1 & 0
heartData.loc[heartData['label'] > 0 , 'label'] = 1
heartData.head()
```

The output is as follows:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0
4	41.0	0.0	2.0	130.0	204.0	0.0	2.0	172.0	0.0	1.4	1.0

Figure 16.26: Change of values in the DataFrame

> **NOTE**
>
> The output has been truncated for presentation purposes.
> The complete output can be found here: https://packt.live/2GbJloz.

6. Drop all **`NA`** values using the **`.dropna()`** function:

```
# Dropping all the rows with na values
newheart = heartData.dropna(axis = 0)
newheart.shape
```

You should get the following output:

```
(297, 14)
```

7. Create the **`Y`** variable using the **`.pop()`** function.

The **`.pop()`** function, as mentioned previously in this chapter, removes the variable in the argument from the DataFrame:

```
# Separating X and y variables
y = newheart.pop('label')
y.shape
```

You should get the following output:

```
(297, )
```

8. Create the **X** variable from the remaining DataFrame.

   Once the target variable is removed, the remaining dataset would be our independent variables:

```
X = newheart
X.head()
```

The output will be as follows:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	6.0
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	3.0
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	7.0
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	3.0
4	41.0	0.0	2.0	130.0	204.0	0.0	2.0	172.0	0.0	1.4	1.0	0.0	3.0

Figure 16.27: Creating the X variable

9. Split the dataset into training and testing sets using **`train_test_split`**:

```
from sklearn.model_selection import train_test_split

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split\
                                   (X, y, test_size=0.3,\
                                    random_state=123)
```

10. Create the necessary processing engine similar to the exercises performed in relation to credit card data.

    In this pipeline, we only include the scaling function since this dataset has only numeric variables:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
numeric_transformer = Pipeline(steps=\
                               [('scaler', StandardScaler())])
numeric_features = X.select_dtypes\
                   (include=['int64', 'float64']).columns
```

```
from sklearn.compose import ColumnTransformer
preprocessor = ColumnTransformer(transformers=\
                                 [('num', numeric_transformer,\
                                   numeric_features)])
```

11. Import the necessary libraries.

    All the library files that are required for this activity are imported as shown here:

```
# Importing necessary libraries
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, \
                             AdaBoostClassifier
```

12. Next, we list the classifiers we are going to try in the spot-checking process:

```
# Creating a list of the classifiers
classifiers = [KNeighborsClassifier(),\
               RandomForestClassifier(random_state=123),\
               AdaBoostClassifier(random_state=123),\
               LogisticRegression(random_state=123)]
```

13. Now, we loop through the classifiers to identify the best model.

    Initiate a **`for`** loop over all the classifiers and then pass the respective classifier into the estimator. Each of the listed classifiers is passed to the estimator to fit the model and the corresponding scores are printed. This step is similar to the one implemented in *Exercise 16.05*, *Step 4*:

```
# Looping through classifiers to get the best model
for classifier in classifiers:
    estimator = Pipeline(steps=\
                         [('preprocessor', preprocessor),\
                          ('dimred', PCA(10)),\
                          ('classifier',classifier)])
    estimator.fit(X_train, y_train)
    print(classifier)
    print("model score: %.2f" % estimator.score(X_test, y_test))
```

You should get something similar to the following output:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
model score: 0.78
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=123,
                       verbose=0, warm_start=False)
model score: 0.80
AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None, learning_rate=1.0,
                   n_estimators=50, random_state=123)
model score: 0.72
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=123, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
model score: 0.80
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning: The default value of
n_estimators will change from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

Figure 16.28: Report for the best model

The preceding output is the report of the scores for each of the classifiers. From the preceding output, we can see the corresponding scores for all the classifiers that have been included in the **`for`** loop. We see that KNN has a score of **`78%`**, random forest has a score of **`80%`**, Adaboost has a score of **`72%`**, and logistic regression has a score of **`80%`**. As logistic regression is one of the classifiers that has given the best result, we select it as the classifier.

14. We'll now create the pipeline using logistic regression, as we did in *Exercise 16.06*.

    A new pipeline is created by stacking together the preprocessor, dimensionality reduction aspect, and logistic regression classifier:

```
# Creating a pipeline with Logistic Regression
pipe = Pipeline(steps=\
                [('preprocessor', preprocessor),\
                 ('dimred', PCA()),('classifier',\
                  LogisticRegression(random_state=123))])
```

15. Let's now define the parameters of the models using a dictionary.

    All the different parameters that need to be experimented with are listed here. This is to initiate the grid search process:

```
# Defining the parameters as a dictionary
param_grid = {'dimred__n_components':[10,11,12,13],\
              'classifier__penalty' : ['l1', 'l2'],\
              'classifier__C' : [1,3, 5],\
              'classifier__solver' : ['liblinear']}
```

16. Define the estimator function, as in *Exercise 16.06*.

    Now, create the estimator function using the **`GridSearchCv`** function. The arguments for the **`GridSearchCV`** function are the pipeline we defined earlier, the number of cross-validation folds, and the dictionary of parameters we want to explore. This is implemented in the following code snippet:

```
from sklearn.model_selection import GridSearchCV
# Fitting the grid search
estimator = GridSearchCV(pipe, cv=10, param_grid=param_grid)
```

17. Next, fit the estimator we created on the training set. As there are multiple parameters to be iterated, this step will be a time-consuming one:

```
# Fitting the estimator on the training set
estimator.fit(X_train,y_train)
```

18. Now, print the best score and parameters, as done in *Exercise 16.06*.

    The best scores and the best set of parameters are printed from the estimator using the **`estimator.best_score_`** argument and **`estimator.best_params_`**:

```
# Printing the best score and best parameters
print("Best: %f using %s" % \
      (estimator.best_score_, estimator.best_params_))
```

    You should get something like the following output:

```
Best: 0.845411 using {'classifier__C': 1, 'classifier__penalty': 'l2', 'classifier__solver': 'liblinear', 'dimred__n_components': 12}
```

Figure 16.29: Output showing the best scores

19. Now, predict using the best estimator.

    The aim of the grid search in the previous step was to find the best combination of parameters. These parameters will be used by the estimator function to predict on the test set:

```
# Predicting with the best estimator
pred = estimator.predict(X_test)
```

20. Let's print the classification report:

```
# Printing the classification report
from sklearn.metrics import classification_report
print(classification_report(pred,y_test))
```

    You should get something like the following output:

```
              precision    recall  f1-score   support

           0       0.86      0.82      0.84        51
           1       0.78      0.82      0.80        39

    accuracy                           0.82        90
   macro avg       0.82      0.82      0.82        90
weighted avg       0.82      0.82      0.82        90
```

Figure 16.30: Classification report for the model

From the results, we can see that with the best parameters that were identified during the grid search process, we improved the results of the logistic regression model from **`0.80`** to **`0.82`**.

From a business perspective, this score of **82%** means that out of the total cases of patient data, we were correctly able to identify **82%** of likely cases of customer heart disease. If we look at the recall values of each class, we will see how the model is faring for each class. From the classification report, we can see that both classes have a recall value of **82%**. This means that for patients with heart disease and without heart disease, the classifier was correctly able to predict **82%** of the available cases in the test set.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/2Q8pr35.
>
> You can also run this example online https://packt.live/31i7be1.

CHAPTER 17: AUTOMATED FEATURE ENGINEERING

ACTIVITY 17.01: BUILDING A CLASSIFICATION MODEL WITH FEATURES THAT HAVE BEEN GENERATED USING FEATURETOOLS

SOLUTION:

1. Open a Colab notebook.
2. Define the path to the GitHub repository:

```
# Defining the path to the GitHub repository
file_url = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter17/Datasets/adult.csv'
```

3. Load the data using **`pandas`**:

```
# Loading data using pandas
import pandas as pd
adultData = pd.read_csv(file_url,sep=",",na_values = "?")
adultData.head()
```

You should get a similar output to the following:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours	native	label
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	0
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	0
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	0
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	0
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	0

Figure 17.31: Loading the data using pandas

4. Now, let's drop all the **`na`** values using the **`dropna()`** function. The **`how = 'any'`** variable drops rows in which you encounter **`na`** values:

```
# Dropping the na values
adultData = adultData.dropna(axis = 0, how = 'any')
adultData.shape
```

You should get a similar output to the following:

```
(30162, 14)
```

5. Remove the target variable. We can do this using the **`.pop()`** function:

```
# Removing the target variable
Y = adultData.pop('label')
```

The **`.pop()`** function removes the defined variable from the dataset.

6. Split the dataset into train and test sets using the **`train_test_split()`** function:

```
from sklearn.model_selection import train_test_split

# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split\
                                   (adultData, Y, \
                                    test_size=0.3,\
                                    random_state=123)
```

7. In this activity, we will use pipelines to scale numerical variables and create dummy variables from categorical variables. This implementation is similar to the exercises we completed in *Chapter 16, Machine Learning Pipelines*:

```
"""
Using pipeline to transform categorical variable
and numeric variables
"""

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder

categorical_transformer = Pipeline(steps=\
                                   [('onehot', OneHotEncoder\
                                    (handle_unknown='ignore'))])
numeric_transformer = Pipeline(steps=[('scaler', \
                               StandardScaler())])
```

First, we define the categorical and numerical transformers, which are one-hot encoding and scaling, respectively.

8. Define the data types for the categorical variables and numerical variables.

   After defining the transformation pipelines, we need to define the categorical and numerical data types. This step is similar to what we did in *Chapter 16, Machine Learning Pipelines*:

```
"""
Defining data types for numeric and categorical features
"""
numeric_features = adultData.select_dtypes\
                   (include=['int64', 'float64']).columns

categorical_features = adultData.select_dtypes\
                       (include=['object']).columns
```

   In the preceding implementation, we select the numerical features and categorical features. The respective features are selected using the **`.adult_dtypes()`** function. *int64* and *float64* are the data types for numerical features, while *object* is the data type for categorical features.

9. In this step, we create the processor pipeline using the **`ColumnTransformer()`** function in scikit learn:

```
# Defining preprocessor
from sklearn.compose import ColumnTransformer
preprocessor = ColumnTransformer\
               (transformers=[('num', numeric_transformer, \
                               numeric_features),\
                              ('cat', categorical_transformer, \
                               categorical_features)])
```

   As seen from the implementation, we give the necessary transformations through the **`transformers`** argument. The first transformer is the numerical transformer, which is represented using the **`numeric`** string. Then, we apply the transformer, **`numTransformer`** , which is the scaling function on the numerical features, **`numFeatures`**. Similarly, we define the appropriate transformations on the categorical variables.

10. Now, create the estimator. Similar to what we did in *Chapter 16, Machine Learning Pipelines*, we create the estimator that contains the processor and logistic regression classifier:

```
"""
Defining the estimator for processing and classification
"""
from sklearn.linear_model import LogisticRegression
estimator = Pipeline(steps=[('preprocessor', preprocessor),\
                            'classifier',\
                            LogisticRegression\
                            (random_state=123))])
```

The estimator is created using the **`Pipeline()`** function. We give the processes that have to be executed in the pipeline as the *steps* argument. In this implementation, the two steps are the preprocessing step and building the classifier using a logistic regression function.

11. Fit the estimator on the training set and then print the model's score:

```
# Fit the estimator on the training set
estimator.fit(X_train, y_train)
print("model score: %.2f" % \
      estimator.score(X_test, y_test))
```

You should get a similar output to the following:

```
Model score: 0.85
```

After the estimator is created, it is fit on the training set using the **`.fit()`** function. The scores of the model on the test set are then printed.

12. Predict on the test set:

```
# Predict on the test set
pred = estimator.predict(X_test)
```

Once the estimator is fit on the training set, we can generate the predictions on the test set using the **`.predict()`** function.

13. Generate the classification report and print it for the predictions that were generated:

```
# Generating classification report
from sklearn.metrics import classification_report
print(classification_report(pred,y_test))
```

You should get a similar output to the following:

```
              precision    recall  f1-score   support

           0       0.93      0.88      0.91      7189
           1       0.62      0.75      0.68      1860

    accuracy                           0.85      9049
   macro avg       0.77      0.81      0.79      9049
weighted avg       0.87      0.85      0.86      9049
```

Figure 17.32: Expected classification matrix

From the preceding output, we can see that the benchmark model has an accuracy of **85%**. We would also be interested in the recall values of the different classes. Class 0 has a recall value of **88%**, which means that out of **7189** adults who did not earn an income of more than **50,000** per year, **88%** were correctly identified. Class **1** has a recall value of **75%**, which indicates that **75%** of adults who earned more than **50,000** per year were correctly identified.

14. Now, create the customer ID for tracking entities.

Similar to *Exercise 17.01*, we will create the parent entity ID. We attach a string called **record** with the index values:

```
# Creating the Ids for parent entity
adultData['parentID'] = adultData.index.values

adultData['parentID'] = 'record' + \
                        adultData['parentID'].astype(str)
```

The created ID is used for tracking the parent ID when the automated features are generated. In the preceding code, we created a new ID called **parentID** by representing the ID name in a square bracket **[]** with the original dataset. A string called **record** is then attached to the index values of the dataset to create unique IDs for each record in the dataset.

15. Create a work class ID. There are seven unique values for the work class. All of these unique values have to be mapped to an ID starting from 1 using the **`.loc()`** function. This is implemented as follows:

```
# Creating unique Ids for entity workclass
adultData.loc[adultData.workclass == \
              ' Federal-gov','workId']= 1
adultData.loc[adultData.workclass == \
              ' Local-gov','workId']= 2
adultData.loc[adultData.workclass == \
              ' Private','workId']= 3
adultData.loc[adultData.workclass == \
              ' Self-emp-inc','workId']= 4
adultData.loc[adultData.workclass == \
              ' Self-emp-not-inc','workId']= 5
adultData.loc[adultData.workclass == \
              ' State-gov','workId']= 6
adultData.loc[adultData.workclass == \
              ' Without-pay','workId']= 7
```

In the preceding code, we specify a condition within the square brackets. For example, the first assignment, **`[adultData.workclass == ' Federal-gov','workId']= 1`**, means that whereever the **`workclass`** variable is equal to the **`' Federal-gov'`** string, the **`workId`** variable has to be assigned a value of **`1`**. All the other commands are similar.

16. Create Occupation IDs. There are **`14`** unique values for Occupation. All of these are mapped to indexes **`1`** to **`14`**, as shown in the following code:

```
# Creating unique IDs for occupation
adultData.loc[adultData.occupation == \
              ' Adm-clerical','occuId']= 1
adultData.loc[adultData.occupation == \
              ' Armed-Forces','occuId']= 2
adultData.loc[adultData.occupation == \
              ' Craft-repair','occuId']= 3
adultData.loc[adultData.occupation == \
              ' Exec-managerial','occuId']= 4
adultData.loc[adultData.occupation == \
              ' Farming-fishing','occuId']= 5
adultData.loc[adultData.occupation == \
              ' Handlers-cleaners','occuId']= 6
```

```
adultData.loc[adultData.occupation == \
              ' Machine-op-inspct','occuId']= 7
adultData.loc[adultData.occupation == \
              ' Other-service','occuId']= 8
adultData.loc[adultData.occupation == \
              ' Priv-house-serv','occuId']= 9
adultData.loc[adultData.occupation == \
              ' Prof-specialty','occuId']= 10
adultData.loc[adultData.occupation == \
              ' Protective-serv','occuId']= 11
adultData.loc[adultData.occupation == \
              ' Sales','occuId']= 12
adultData.loc[adultData.occupation == \
              ' Tech-support','occuId']= 13
adultData.loc[adultData.occupation == \
              ' Transport-moving','occuId']= 14
```

This implementation is similar to the one we executed in the previous step. In this step, the **`occuId`** variable is updated with the respective value based on the string value in the **`occupation`** variable.

17. Now, we will import the library packages that we need in order to create features:

```
# Importing necessary libraries
import featuretools as ft
import numpy as np
```

18. Create the parent entity. The parent entity is created using the **`.Entityset()`** function:

```
# creating the entity set 'adultentities'
adultentities = ft.EntitySet(id = 'Adult')
```

In the preceding implementation, we define a string called **`Adult`** as the name of the entity set.

19. Parent entities are mapped to the data frame using the **`entity_from_dataframe()`** function:

```
"""
Mapping a dataframe to the entityset to form the
parent entity
"""
```

```
adultentities.entity_from_dataframe\
              (entity_id = 'Parent Data', \
               dataframe = adultData, \
               index = 'parentID')
```

Once the entity set has been created, the first step is to create the parent entity and then map the data frame to the entity set. The index for tracking the parent entity is **`parentID`**.

You should get the following output:

```
Entityset: Adult
  Entities:
    Parent Data [Rows: 30162, Columns: 16]
  Relationships:
    No relationships
```

Figure 17.33: Mapping the parent entity to the dataset

From the preceding output, we can see that the Parent entity has been created, and it has **`30162`** rows and **`16`** columns. We can see that no relationships with the other entities have been created so far; these will be created in the next step.

20. Now, map all the entities and set the relationships.

    In this step, we'll map all the entities and set the relationships using the **`.normalize_entity()`** function. Please note that for the education entity, we have not created any IDs since the education-num variable has a mapping to all the unique values of the education variable:

```
# Mapping to parent entity and setting the relationship
adultentities.normalize_entity\
              (base_entity_id='Parent Data', \
               new_entity_id='education', \
               index = 'education-num', \
               additional_variables = ['education'])

adultentities.normalize_entity\
              (base_entity_id='Parent Data', \
               new_entity_id='Workclass', \
               index = 'workId', \
               additional_variables = ['workclass'])
```

```
adultentities.normalize_entity\
               (base_entity_id='Parent Data', \
                new_entity_id='Occupation', \
                index = 'occuId', \
                additional_variables = ['occupation'])
```

In this implementation, we give the parent entity to the **base_entity** argument and the child entities to the **new_entity_id** argument. We also define the index of the child entity, which has to be used for tracking. In addition, we give the variable name that is related to the child entity in the **additional_variables** argument.

You should get the following output:

```
Entityset: Adult
  Entities:
    Parent Data [Rows: 30162, Columns: 13]
    education [Rows: 16, Columns: 2]
    Workclass [Rows: 7, Columns: 2]
    Occupation [Rows: 14, Columns: 2]
  Relationships:
    Parent Data.education-num -> education.education-num
    Parent Data.workId -> Workclass.workId
    Parent Data.occuId -> Occupation.occuId
```

Figure 17.34: Mapping all the entities to the relationships in the dataset

From the preceding output, we can see that all the child entities (**education**, **Workclass**, and **Occupation**) have been created. We can also see the relationship that is created between the parent entity and the child entities.

21. Create the aggregation and transformation primitives, as shown in the following code snippet:

```
# Creating aggregation and transformation primitives
aggPrimitives=['std', 'min', 'max', 'mean', 'last', 'count']
tranPrimitives=['percentile', 'subtract', 'divide']
```

In the preceding implementation, we are configuring the aggregation and transformation primitives. We are adding the required primitives such as standard deviation (**std**), **min** , percentile, and so on to the respective primitive types. Once the primitives have been configured and defined separately, they override the default primitives.

22. Define the DFS with the created primitives.

    In this step, we define the DFS with the created primitives. We set the depth to **2**:

```
# Defining the new set of features
feature_set, feature_names = ft.dfs\
                              (entityset=adultentities, \
                               target_entity = 'Parent Data',\
                               agg_primitives=aggPrimitives, \
                               trans_primitives=tranPrimitives,\
                               max_depth = 2, verbose = 1,\
                               n_jobs = 1)
```

    In the preceding implementation, we define deep feature synthesis using the **ft.dfs()** function. We give the name of the entity set under the **entityset** argument and the parent entity name under the **target_entity** argument. We also define the primitives we configured in the previous step. **max_depth** defines how deep the stacking of variables has to be implemented.

    You should get a similar output to the following:

```
Built 1076 features
Elapsed: 00:31 | Remaining: 00:00 | Progress: 100%|██████████| Calculated: 11/11 chunks
```

Figure 17.35: Output showing the number of features that were created

    From the preceding output, we can see that **1076** features have been created.

23. Once the feature sets have been created, the indexing will be all jumbled up. We need to reindex them so that the index is similar to the original dataset. This is implemented as follows:

```
# Reindexing the feature_set
feature_set = feature_set.reindex(index=adultData['parentID'])
feature_set = feature_set.reset_index()
```

    In the first line, reindexing is done using the **.reindex()** function. As an argument, we give the target index, based on which the reindexing has to be done. In the argument, we specify that the indexing has to be done based on the order of **parentID**. Once this line is implemented, the index of the data frame becomes 'record01','record02' ..., and so on. The second line, that is, **.reset_index()**, is used to change this index to 0, 1, 2, and so on.

24. After the feature set has been created, we need to print the shape of the feature set:

```
# Displaying the feature set
feature_set.shape
```

You should get the following output:

```
(30162, 1077)
```

From the preceding output, we can see that the new dataset, which has **1077** new features, has been created. The number of rows, that is, **30162**, remains the same.

25. In the feature set, there will be some features that are related to the IDs that we created. These aren't necessary for modeling. Therefore, we can remove them. We can do this as follows:

```
# Dropping all Ids
X = feature_set[feature_set.columns[~feature_set\
                .columns.str.contains('parentID\
                |education-num|workId|occuId')]]
```

In this implementation, the tilde **~** sign means negation. Here, we are subsetting the feature set with those features that don't contain **parentID**, **education-num** , **workId**, or **occuId**.

26. Replace all the infinity values with **nan** values.

One after-effect of a transformation primitive such as divide is to create features with infinity values. This happens when there are features that contain 0. As you know, division with 0 will generate an infinity value. These infinity values have to be removed from the data frame. This is done by replacing all the infinity values with **nan** values and then dropping the nan values. The first step is implemented as follows using the **.replace()** function:

```
# Replacing all columns with infinity with nan
X = X.replace([np.inf, -np.inf], np.nan)
```

Here, **np**. **Inf**, and **–np.inf** stand for infinity values.

27. Once we've replaced the infinity values with **nan**, these columns can be dropped from the dataset. This is implemented using the **.dropna()** function:

```
# Dropping all columns with nan
X = X.dropna(axis=1, how='any')
X.shape
```

You should get the following output:

```
(30162, 893)
```

28. In the preceding implementation, **axis = 1** means along the columns. **how = 'any'** means drop any column contacting nan values. We can see that the number of features drops from **1077** to **893** after removing all the redundant columns.

    Now, let's split the new dataset into train and test sets for modeling using the **train_test_split()** function:

```
# Splitting train and test sets
from sklearn.model_selection import train_test_split

# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split\
                                   (X, Y, test_size=0.3,\
                                    random_state=123)
```

29. Like we did in the benchmark model creation step, let's create the processing step:

```
# Creating the preprocessing pipeline
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder

categorical_transformer = Pipeline(steps=\
                                   [('onehot',\
                                     OneHotEncoder\
                                     (handle_unknown='ignore'))])

numeric_transformer = Pipeline\
                      (steps=[('scaler', StandardScaler())])

numeric_features = X.select_dtypes\
                   (include=['int64', 'float64']).columns
```

```
categorical_features = X.select_dtypes\
                       (include=['object']).columns

from sklearn.compose import ColumnTransformer
preprocessor = ColumnTransformer\
               (transformers=[('num', numeric_transformer, \
                               numeric_features),\
                              ('cat', categorical_transformer,\
                               categorical_features)])
```

The creation of the pipeline for transforming the numeric and categorical variables using the **`ColumnTransformer()`** function is the same as what we implemented in the benchmark model for the same dataset.

30. Let's create the estimator function, which contains the preprocessing step and the classifier layer. After this, we'll fit the estimator on the training set and print the scores:

```
"""
Creating the estimator function and fitting
the training set
"""
estimator = Pipeline(steps=\
                     [('preprocessor', preprocessor), \
                      ('classifier', \
                       LogisticRegression(random_state=123))])
estimator.fit(X_train, y_train)
print("model score: %.2f" % \
      estimator.score(X_test, y_test))
```

You should get the following output:

```
model score: 0.86
```

As seen from the preceding output, the accuracy level has improved from **`85%`** to **`86%`** using the new dataset. Let's see what the classification report looks like.

31. Predict on the test set:

```
# Predicting on the test set
pred = estimator.predict(X_test)
```

After fitting the estimator on the train set, we generate the predictions by using the **`predict()`** function on the test set.

32. Once the predictions have been generated, we can print the classification report:

```
# Generating the classification report
from sklearn.metrics import classification_report

print(classification_report(pred,y_test))
```

You should get a similar output to the image below:

```
              precision    recall  f1-score   support

           0       0.93      0.89      0.91      7134
           1       0.64      0.76      0.69      1915

    accuracy                           0.86      9049
   macro avg       0.79      0.82      0.80      9049
weighted avg       0.87      0.86      0.86      9049
```

Figure 17.36: Expected classification matrix

From the preceding output, we can see that the accuracy scores have improved from **85%** to **86%**. There is also an improvement in the precision, recall, and f1-score of the minority class (yes). All of these values have increased from **62%**, **75%**, and **68%** to **64%**, **76%**, and **69%**, respectively.

From a business perspective, the result indicates that out of the total **9,049** adults, **86%** of them have been correctly identified as earning more than **50,000** per year or not.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/3gibwBY.
>
> This section does not currently have an online interactive example, but can be run as usual on Google Colab.

CHAPTER 18: MODEL AS A SERVICE WITH FLASK

ACTIVITY 18.01: TRAIN AND DEPLOY AN INCOME PREDICTOR MODEL USING FLASK

SOLUTION

1. Open a new Colab notebook.
2. Import the **`pandas`**, **`pickle`**, **`joblib`**, and **`RandomForestClassifier`** packages from **`sklearn.ensemble`**, as well as **`train_test_split`** from **`sklearn.model_selection`**:

```
import pandas as pd
import joblib
import pickle
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
```

3. Assign the link to the dataset to a variable called **`file_url`**:

```
file_url = 'https://raw.githubusercontent.com'\
           '/PacktWorkshops/The-Data-Science-Workshop'\
           '/master/Chapter18/Dataset/phpMawTba.csv'
```

4. Load the dataset into a DataFrame using **`pd.read_csv()`**:

```
df = pd.read_csv(file_url)
```

5. Print out the first five rows of this DataFrame

```
df.head()
```

You should get the following output:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct
4	18	?	103497	Some-college	10	Never-married	?

Figure 18.47: First five rows of the dataset

6. Extract the **`'class'`** response variable using the **`.pop()`** method and save it into a variable called **`y`**:

```
y = df.pop('class')
```

7. Create a list called **`cat_columns`** containing only the columns of type **`'object'`** using the **`dtype`** attribute and print its content:

```
cat_columns = [col for col in df.columns \
               if df[col].dtype == 'object']
cat_columns
```

You should get the following output:

```
['workclass',
 'education',
 'marital-status',
 'occupation',
 'relationship',
 'sex',
 'native-country']
```

Figure 18.48: List of categorical variables

8. Split the **df** and **y** DataFrames into training and test sets using the **train_test_split** function with the parameters **test_size=0.33** and **random_state=8**:

```
X_train, X_test, y_train, y_test = train_test_split\
                                   (df, y, test_size=0.33, \
                                    random_state=8)
```

9. Create an empty dictionary called **column_categories**:

```
column_categories = {}
```

10. Iterate through **cat_columns** and populate the dictionary with the column name and the list of categories using the **.astype()** method and the **.cat.categories** attribute:

```
for col in cat_columns:
    column_categories[col] = X_train[col].astype('category')\
                                          .cat.categories
```

11. Save **column_categories** and **cat_columns** into files called **categories_data.pkl** and **categorical_columns.pkl** respectively using the **pickle.dump()** method:

```
pickle.dump(column_categories, \
            open("categories_data.pkl", "wb"))
pickle.dump(cat_columns, \
            open("categorical_columns.pkl", "wb"))
```

12. Create a function called **apply_categories** that takes a DataFrame and a dictionary as inputs and will import **CategoricalDtype** from **pandas.api.types**, iterate through this dictionary, and convert each column (keys) with the list of categories (values) using the **.astype()** method and **CategoricalDtype**:

```
def apply_categories(input_df, cat_dict):
    from pandas.api.types import CategoricalDtype
    for col, cat in cat_dict.items():
        input_df[col] = input_df[col].astype\
                        (CategoricalDtype(categories=cat))
    return input_df
```

13. Apply this function on **`X_train`** and **`column_categories`** and save the result in a new DataFrame called **`X_train_cat`**. Print the data type of its columns using the **`.dtypes`** attribute:

```
X_train_cat = apply_categories(X_train, column_categories)
X_train_cat.dtypes
```

You should get the following output:

```
age                int64
workclass       category
fnlwgt             int64
education       category
education-num      int64
marital-status  category
occupation      category
relationship    category
sex             category
capital-gain       int64
capital-loss       int64
hours-per-week     int64
native-country  category
dtype: object
```

Figure 18.49: Data type of each column

14. Perform one-hot encoding on the categorical columns using the **`.get_dummies()`** method and save the result into a new variable called **`X_train_final`**:

```
X_train_final = pd.get_dummies(X_train_cat, columns=cat_columns)
```

15. Instantiate a **`RandomForestClassifier`** with **`random_state=8`** and train it with the training sets using the **`.fit()`** method. Save the model into a file called **`model.pkl`** using the **`joblib.dump()`** method:

```
rf_model = RandomForestClassifier(random_state=8)
rf_model.fit(X_train_final, y_train)
joblib.dump(rf_model, "model.pkl")
```

16. Import the **socket**, **threading**, **requests**, **json**, and **numpy** packages, the **Flask** class, and the **jsonify** and **request** functions from the **flask** package:

```
import socket
import threading
import requests
import json
from flask import Flask, jsonify, request
import numpy as np
```

17. Create a new **Flask** app and save it into a variable called **app**:

```
app = Flask(__name__)
```

18. Load the pre-trained model from the **model.pkl** file using **joblib.load()** and save it into a variable called **trained_model**. Load the saved dictionary from **categories_data.pkl** using **pickle.load()** and save it into a variable called **var_means**:

```
trained_model = joblib.load("model.pkl")
var_means = pickle.load(open("categories_data.pkl", "rb"))
cat_cols = pickle.load(open("categorical_columns.pkl", "rb"))
```

19. Create an API endpoint for the **api** path that accepts only POST requests and will call a function called **predict()**. This function will read the JSON received using the **request.get_json()** method, transform it into a DataFrame, apply the **apply_categories()** function on it with **var_means**, perform one-hot encoding with **.get_dummies()**, predict the outcome with **trained_model**, convert the prediction from a **numpy** array to a string with **array2string()**, and then convert to JSON with **jsonify()**:

```
@app.route('/api', methods=['POST'])
def predict():
    data = request.get_json()
    df_test = pd.DataFrame(data, index=[0])
    df_test_clean = apply_categories(df_test, \
                                     var_means)
    df_test_final = pd.get_dummies(df_test_clean, \
                                   columns=cat_cols)
    prediction = trained_model.predict(df_test_final)
    str_pred = np.array2string(prediction)
    return jsonify(str_pred)
```

20. Create a new thread for running your Flask app using the **`threading.Thread`** method with the following parameters: **`target=app.run, kwargs={'host':'0.0.0.0','port':80}`**:

```
flask_thread = threading.Thread(target=app.run, \
                                kwargs={'host':'0.0.0.0', \
                                'port':80})
flask_thread.start()
```

You should get the following output:

```
 * Serving Flask app "__main__" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Figure 18.50: Log of the Flask app

21. Select the first record of **`X_test`** and convert it into JSON format using the **`.to_json()`** method:

```
record = X_test.iloc[0,].to_json()
record
```

You should get the following output:

```
'{"age":51,"workclass":" Private","fnlwgt":106151,"education":" 11th","educa
```

Figure 18.51: Record in JSON format

22. Create a dictionary called **`headers`** with the following key-value pairs: **`'content-type'`**: **`'application/json'`**, **`'Accept-Charset'`**: **`'UTF-8'`**. Extract into a new variable called **`ip_address`** the IP address of the host using the **`socket.gethostname()`** and **`socket.gethostbyname()`** methods:

```
headers = {'content-type': 'application/json', \
           'Accept-Charset': 'UTF-8'}
ip_address = socket.gethostbyname(socket.gethostname())
```

23. Send an HTTP POST request to the server using the **`requests.post()`** method with the HTTP URL to the endpoint, using **`record`** and **`headers`** as its parameters, and print its **`.text`** attribute:

```
r = requests.post(f"http://{ip_address}/api", \
                  data=record, headers=headers)
r.text
```

You should get the following output:

```
172.28.0.2 - - [06/Nov/2019 11:22:42] "POST /api HTTP/1.1" 200 -
'"[\' <=50K\']"\n'
```

Figure 18.52: Log and prediction of the POST request

From the output, we observe that the **`POST`** request was successful: the server returned the code 200. We received the prediction from the model for the record we sent, and it has predicted the person has an income below the 50k mark.

> **NOTE**
>
> To access the source code for this specific section, please refer to https://packt.live/32Ge2NR.
>
> This section does not currently have an online interactive example, but can be run as usual on Google Colab.