APPENDIX

ACTIVITY 16.01: COMPLETE ML WORKFLOW IN A PIPELINE

Solution:

1. Open a new Colab notebook
2. Load the dataset from the GitHub repository:

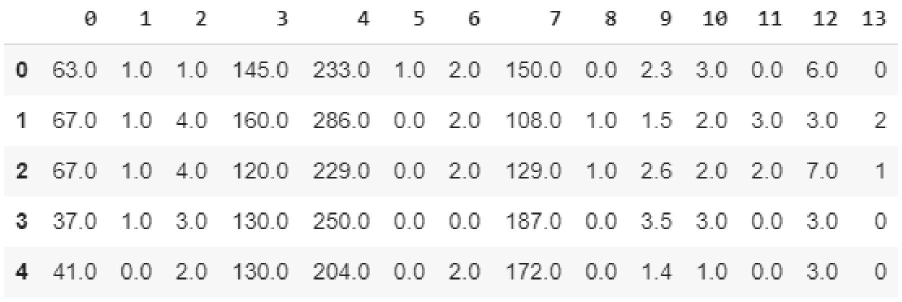
> **NOTE**
>
> The dataset to be used in this activity can be found on our GitHub repository at https://raw.githubusercontent.com/PacktWorkshops/The-Data-Science-Workshop/master/Chapter16/Dataset/processed.cleveland.data

```
#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:



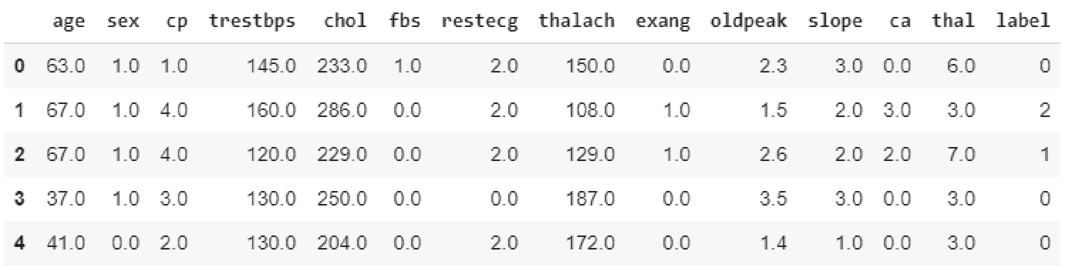
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	6.0	0
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	3.0	2
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	7.0	1
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	187.0	0.0	3.5	3.0	0.0	3.0	0
4	41.0	0.0	2.0	130.0	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:

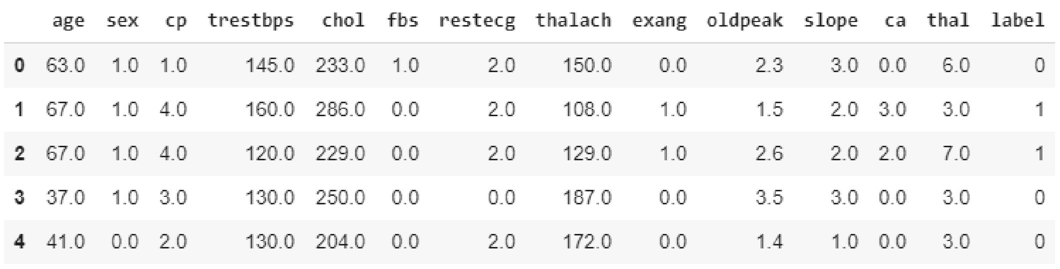


	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	label
0	63.0	1.0	1.0	145.0	233.0	1.0	2.0	150.0	0.0	2.3	3.0	0.0	6.0	0
1	67.0	1.0	4.0	160.0	286.0	0.0	2.0	108.0	1.0	1.5	2.0	3.0	3.0	2
2	67.0	1.0	4.0	120.0	229.0	0.0	2.0	129.0	1.0	2.6	2.0	2.0	7.0	1
3	37.0	1.0	3.0	130.0	250.0	0.0	0.0	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()
```



	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	1
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 16.26: Change of values in the DataFrame

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:

```
# Seperating X and y variables
y = newheart.pop('label')
y.shape
```

You should get the following output:

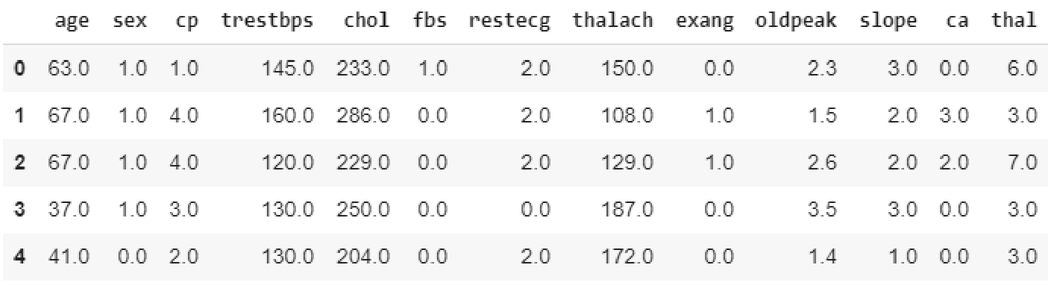
```
(297, )
```

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:

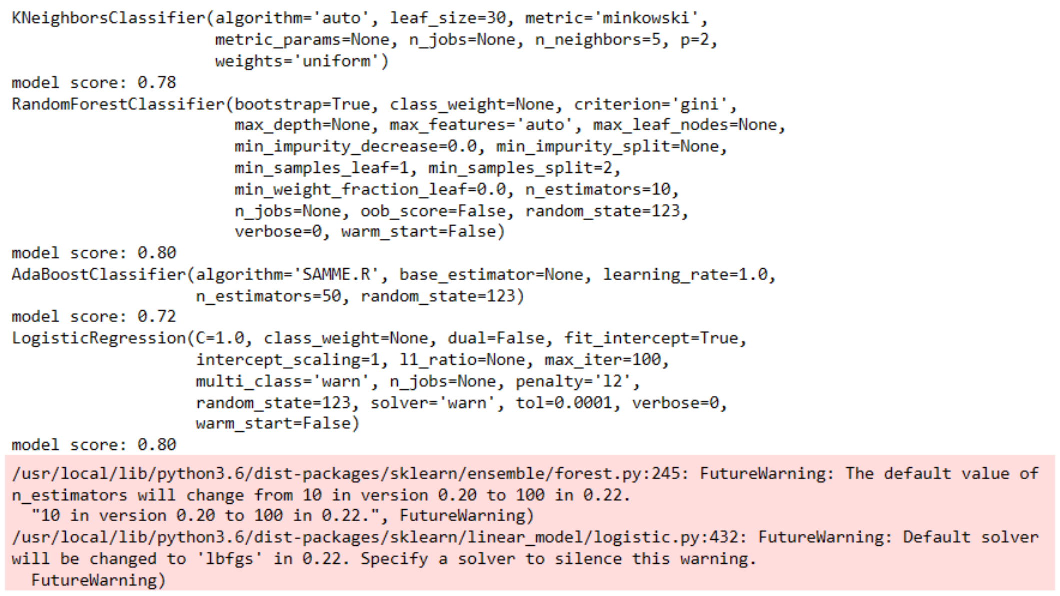
```
# Creating a list of the classifiers
classifiers = [
    KNeighborsClassifier(),
    RandomForestClassifier(random_state=123),
    AdaBoostClassifier(random_state=123),
    LogisticRegression(random_state=123)
    ]
```

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 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:

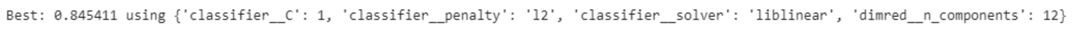
```
# Fitting the estimator on the training set
estimator.fit(X_train,y_train)
```

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 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.

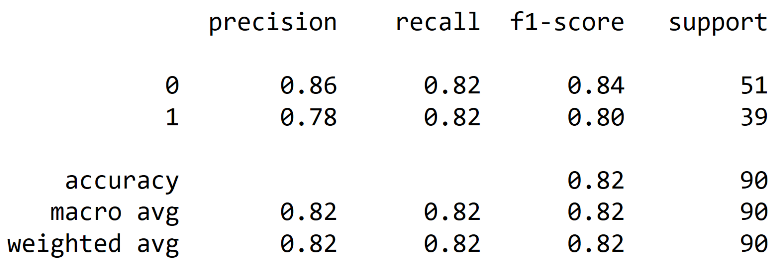
    The aim of the grid search in the previous step was to find the best combination of parameters. These parameters will be used by the estimator function to predict on the test set:

```
# 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 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.

ACTIVITY 17.01: BUILDING A CLASSIFICATION MODEL WITH FEATURES THAT HAVE BEEN GENERATED USING FEATURETOOLS

Solution:

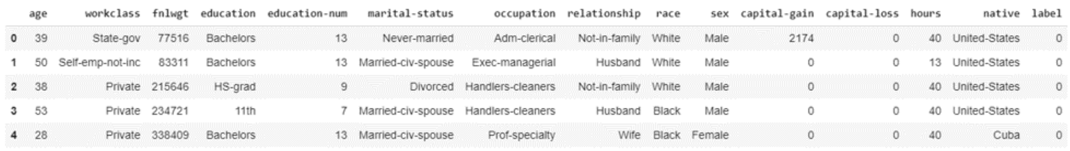
1. Open a Colab notebook.
2. Define the path to the GitHub repository:

```
# 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:

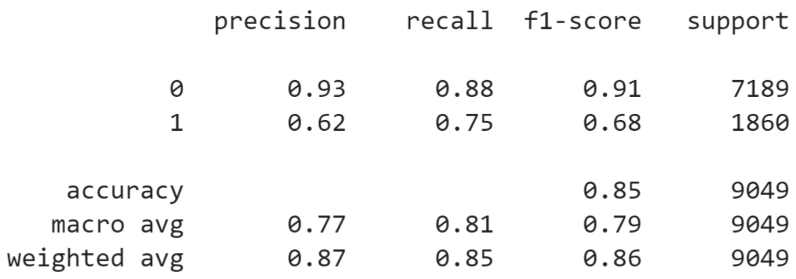
```
# Predict on the test set
pred = estimator.predict(X_test)
```

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')
```

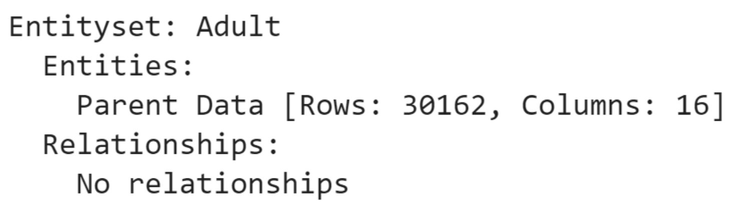
In the preceding implementation, we define a string called **`Adult`** as the name of the entity set.

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.

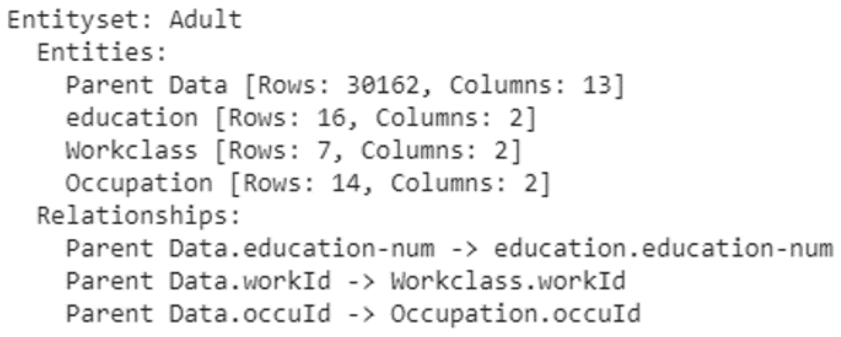
20. Now, map all the entities and set the relationships.

    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.

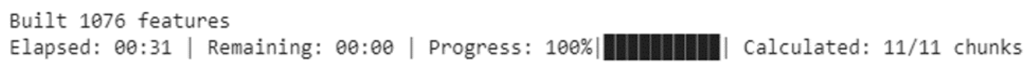
22. Define the DFS with the created primitives.

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:

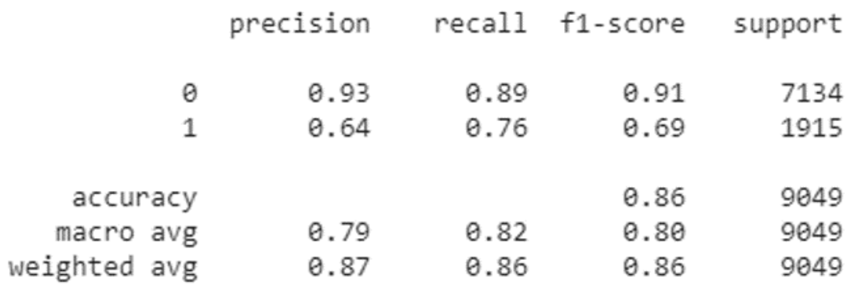
```
# Predicting on the test set
pred = estimator.predict(X_test)
```

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 the following output:



```
              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.

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 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'
```

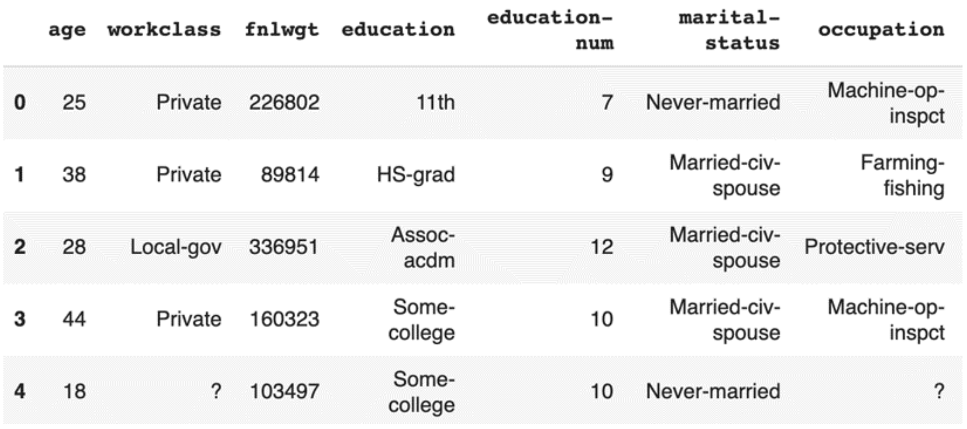
4. Load the dataset into a DataFrame using **`pd.read_csv()`**:

```
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

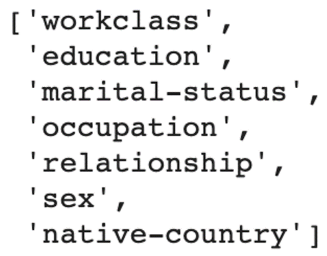
6. Extract the **'class'** response variable using the **.pop()** method and save it into a variable called **y**:

```
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"))
```

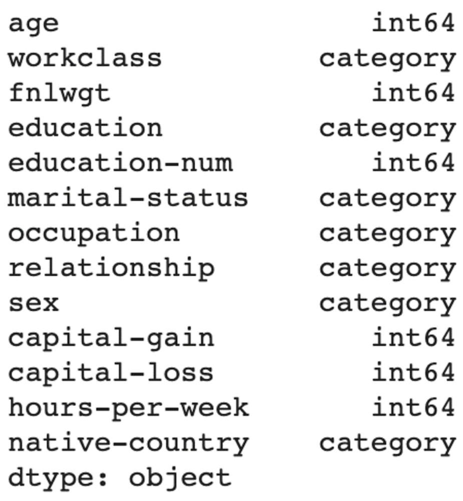
12. Create a function called **apply_categories** that takes a DataFrame and a dictionary as inputs and will import **CategoricalDtype** from **pandas.api.types**, iterate through this dictionary, and convert each column (keys) with the list of categories (values) using the **.astype()** method and **CategoricalDtype**:

```
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"))
```

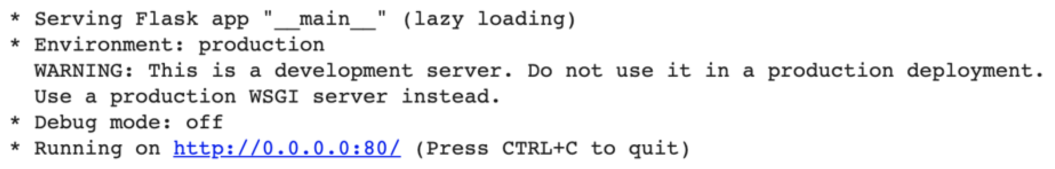
19. Create an API endpoint for the **`api`** path that accepts only POST requests and will call a function called **`predict()`**. This function will read the JSON received using the **`request.get_json()`** method, transform it into a DataFrame, apply the **`apply_categories()`** function on it with **`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:



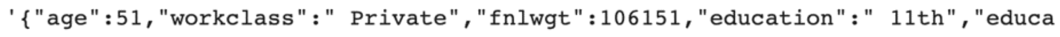
```
 * Serving Flask app "__main__" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

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

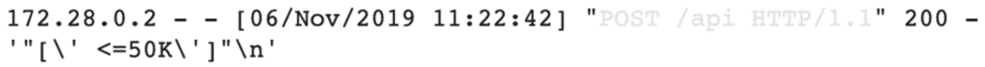
22. Create a dictionary called **`headers`** with the following key-value pairs: **`'content-type'`**: **`'application/json'`**, **`'Accept-Charset'`**: **`'UTF-8'`**. Extract into a new variable called **`ip_address`** the IP address of the host using the **`socket.gethostname()`** and **`socket.gethostbyname()`** 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.