18

MODEL AS A SERVICE WITH FLASK

OVERVIEW

This chapter will teach you how to build a Flask application and create API endpoints for accessing or updating resources from a server. You will also learn how to store and load trained machine learning models and data processing artifacts. You will be able to facilitate the sending of prediction requests for a machine learning model.

By the end of the chapter, you will be able to deploy a machine learning model as a web application for online prediction.

INTRODUCTION

In the previous chapter, we learned how to use a very interesting package for automated feature engineering called **`Featuretools`**. If used properly, it can save you a lot of time and help you to focus on other parts of a project. This chapter will introduce you to another important topic in a data science project: model deployment. This step is optional, depending on the business requirement. However, if we are looking at a model that will be deployed, then we have to consider a few things before putting it into production.

Firstly, we have to consider the business stakeholders. You need to be very confident in your model's performance and its output results. You don't want to put into production a model that is not good enough from the business point of view. Cost may be a second factor to consider. Putting a model into production requires some investment in infrastructure: a server (local or in the cloud) will be needed to host your model. So, there will be a recurring cost involved. Also, you will have to define the process to manage the life cycle of your model. How will you check that your model is not deviating from the ground truth after a while? Will you retrain and update your model every week, or month? How will you keep track of the different versions of your model and the data used for training and testing? These are the kinds of questions you will have to think through and plan for with the different stakeholders involved.

Once you get all the green lights, then you can move your machine learning model into production. This chapter will show you how you can serve your model as an on-demand service via a web API.

BUILDING A FLASK WEB API

Building a web API is the most popular way to expose a machine learning model in a production environment. But before diving into this, we need to understand what a web API is first.

A web API (also known as a web service) is a programming interface that allows web communication between a client and a server. It is usually composed of one or multiple endpoints that expose resources from the server side that can be accessed externally. A web API relies on a request-response messaging mechanism for handling received requests and sent responses.

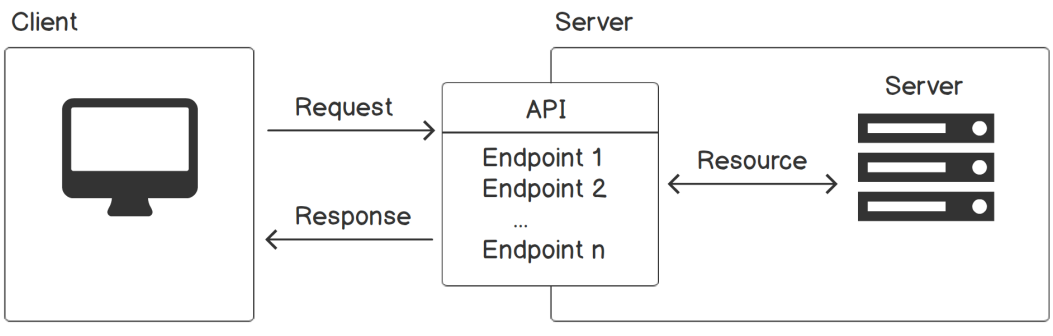

Figure 18.1: Web API schema

A client that wants to access some resources from a server will send an HTTP request to a specific endpoint from the server environment. This endpoint will start the processing of this request and send the result back as an HTTP response.

There are multiple frameworks on the market for building web applications in Python, such as Django, Web2Py, and TurboGears. We will be using Flask in this chapter. It is a very popular framework used for building microservices due to its simplicity and flexibility. We will continue using Google Colab as our main IDE, so the process of building a web app via a notebook will be slightly different than via a normal Python script, but not very different.

Let's see how we can build a very simple web app that displays some text on Google Colab. First, we will focus on the server side. We will import the **`socket`** module and print out the hostname using the **`.gethostname()`** method. We will use this host as our web app server, as you can see in the following code snippet:

```
import socket
hostname = socket.gethostname()
hostname
```

The hostname will be displayed as follows:

```
'c88706aa5893'
```

Figure 18.2: Hostname

> **NOTE**
>
> The hostname may be different in your notebook as your notebook will run on a different server.

We want to get the IP address from this hostname by calling the **`.gethostbyname()`** method with the hostname as a parameter, as you can see in the following code:

```
ip_address = socket.gethostbyname(hostname)
ip_address
```

The IP address will be displayed as follows:

```
'172.28.0.2'
```

Figure 18.3: Host IP address

Now that we have obtained the IP address of the server, we can create our first Flask application using the **`flask`** package. We need to instantiate a Flask object and provide the name of the current module with **`__name__`** as a parameter:

```
from flask import Flask
app = Flask(__name__)
```

Now that we have created an empty Flask web app, let's create an endpoint by adding an **`@app.route()`** decorator to a function called **`hello()`** that will return the text **`Hello Packt!`**:

```
@app.route("/")
def hello():
    return "Hello Packt!"
```

A Python decorator enables us to change the behavior of a function from an existing object. In our example, we are extending the **`hello()`** function as an endpoint of the Flask web application **`app`** using the **`@app.route()`** decorator. We can specify the routing (or URL path) as a parameter of this decorator. Here, we are specifying the home page (or root) with **`/`**, so if we send a request to the IP address **`http://172.28.0.2`**, the Flask application will automatically call the **`hello()`** function.

But before moving on to the client side, we need to launch the Flask app. Usually, to start a Flask application, you just have to call this:

```
app.run()
```

But as we are using Google Colab, we need to create a new thread (another task that can be run in parallel) using the **`Thread`** class from the **`threading`** package. We need to specify the following parameters:

- **`target=app.run`**: This new thread will call the **`app.run()`** method.
- **`kwargs={'host':'0.0.0.0','port':80}`**: This will specify the IP address of the server (**`0.0.0.0`** stands for localhost) and the port used, **`80`** (used for HTTP communication):

```
import threading
flask_thread = threading.Thread(target=app.run, kwargs={'host':'0.0.0.
0','port':80})
```

We can now launch the thread with the **`.start()`** method:

```
flask_thread.start()
```

The output will be as follows:

```
 * 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 18.4: Log after starting the Flask app

Our first Flask application is running now. It is time to test it by simulating a request from the client side. We will send a GET request, which is a type of HTTP request, to the host (**`http://172.28.0.2`**):

```
import requests
r = requests.get("http://172.28.0.2")
```

The output will be as follows:

```
172.28.0.2 - - [01/Nov/2019 10:40:03] "GET / HTTP/1.1" 200 -
```

Figure 18.5: Log of the GET request

This log implies that we called the server with IP address **`172.28.0.2`** on a specific datetime (1st of November 2019 at 10:40), we sent an **`HTTP GET`** request (version 1.1), and the returned status code is **`200`**, which means it was successful.

Now we can look at the response received from this request using the **`.text`** attribute:

```
r.text
```

The output will be as follows:

```
'Hello Packt!'
```

Figure 18.6: HTTP response of the GET request

This is it! We just created our first Flask application with a single endpoint that returns some text. It is as simple as that.

ADDING A POST API ENDPOINT

Now that we have built the overall Flask app, we will now look at adding more complex API endpoints. In the previous section, we learned how to process an HTTP GET request. There are actually other types of HTTP request methods. The two most popular ones are GET and POST. The GET method is used for requesting access to some resources from a server, while the POST method is for sending data to a server to update or create a resource:

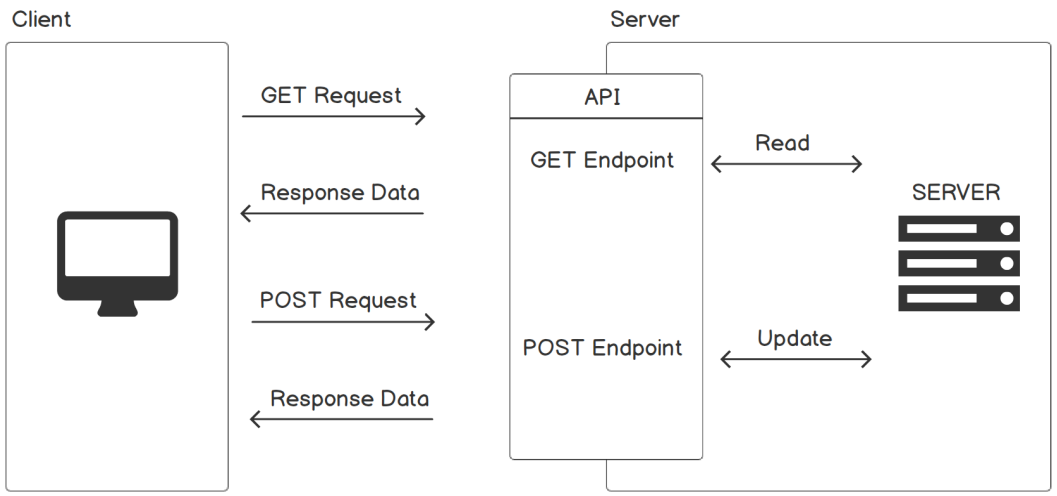

Figure 18.7: GET versus POST requests

Let's try to send a POST request to our API using the **`.post()`** method:

```
r = requests.post("http://172.28.0.2")
```

The output will be as follows:

```
172.28.0.2 - - [01/Nov/2019 22:46:14] "POST / HTTP/1.1" 405 -
```

Figure 18.8: Log of the POST request

This time, the log returns a status code of **`405`**, which means there is an error from the client side stating that the **`POST`** method is not allowed.

This is because when we declared the root endpoint, we didn't specify any HTTP request type, so by default Flask allows only GET requests. To specify which request type an endpoint can process, we just need to provide the list of request types to the **`methods`** parameter of the **`app.route()`** decorator. Let's create a new endpoint that accepts both GET and POST requests:

```
@app.route('/foo', methods=['GET', 'POST'])
def foo():
return "Hello Foo!"
```

Now let's try to send a GET request to this new endpoint:

```
r_get = requests.get("http://172.28.0.2/foo")
r_get.text
```

The output will be as follows:

```
172.28.0.2 - - [01/Nov/2019 23:07:19] "GET /foo HTTP/1.1" 200 -
'Hello Foo!'
```

Figure 18.9: Log and result of the GET request to the foo/ endpoint

Let's do the same but with a POST request:

```
r_post = requests.post("http://172.28.0.2/foo")
r_post.text
```

The output will be as follows:

```
172.28.0.2 - - [01/Nov/2019 23:07:14] "POST /foo HTTP/1.1" 200 -
'Hello Foo!'
```

Figure 18.10: Log and result of the POST request to the foo/ endpoint

Great! Now the POST request is working. At the moment, it is just behaving like a GET request. Let's send some data to the endpoint. But first, let's add a new endpoint that only accepts POST requests and returns the same data it receives. In the endpoint function, we need to read the data from the request using the **`request.get_json()`** method from Flask. This method will return a dictionary. HTTP requests need to return HTTP-compatible text format. The most popular one is JSON (JSON was mentioned in the first chapter). It is very similar to a Python dictionary. The **`jsonify`** function from Flask will convert a Python dictionary to JSON:

```
from flask import jsonify, request

@app.route('/display', methods=['POST'])
def print_item():
  data = request.get_json()
  return jsonify(data)
```

Now we have to send data with the POST request. First, let's create a list of data we will send to this endpoint:

```
data = ['Australia', 'France', 'China']
```

We need to convert it into JSON. We will use the **`.dumps()`** method from the **`json`** package:

```
import json
j_data = json.dumps(data)
```

To send data via a POST request, we need to provide the type of information sent. We will provide a dictionary to the **`header`** parameter of the POST request: **`headers = {'content-type': 'application/json', 'Accept-Charset': 'UTF-8'}`**. This dictionary specifies the type of data. Here it will be JSON, and the text encoding is **`UTF-8`**:

```
headers = {'content-type': 'application/json', 'Accept-Charset': 'UTF-8'}

r = requests.post("http://172.28.0.2/display", data=j_data,
headers=headers)
```

The output will be as follows:

```
172.28.0.2 - - [02/Nov/2019 22:31:40] "POST /display HTTP/1.1" 200 -
```

Figure 18.11: Log of the POST request to the display/ endpoint

Our POST request to the **display/** endpoint was successful, as shown by the logs. Now let's print the response received from the server by looking at the **.text** attribute:

```
r.text
```

The output will be as follows:

```
'["Australia","France","China"]\n'
```

Figure 18.12: HTTP response of the POST request

Great! This is exactly the result we expected from this API POST endpoint. It returned the exact same data it received as input.

EXERCISE 18.01: CREATING A FLASK API WITH ENDPOINTS

In this exercise, we will learn how to create a Flask API with two different endpoints:

- The root endpoint, which will display a welcoming message for any GET request received
- Another one, which will check whether the input data received is empty or not

The following steps will help you complete the exercise:

1. Open a new Colab notebook.
2. Import the **socket**, **threading**, **requests** and **json** 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
```

3. Save the host IP address into a new variable called **ip_address** using the **.gethostbyname()** and **.gethostname()** methods. Display the value of this new variable:

```
ip_address = socket.gethostbyname(socket.gethostname())
ip_address
```

You should get the following output:

```
'172.28.0.2'
```

Figure 18.13: IP address of the host server

4. Create a Flask app and save it into a new variable called **`app`**:

```
app = Flask(__name__)
```

5. Create an API endpoint for the root directory using the **`@app.route()`** decorator that will call a function named **`welcome()`**, which will return the following message: **`Welcome to my API!`**:

```
@app.route("/")
def welcome():
    return "Welcome to my API!"
```

6. 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)
```

Figure 18.14: Log displayed once the Flask app is started

7. Send a HTTP GET request to the server using the **`requests.get()`** method with the HTTP URL of the host IP address and print its **`.text`** attribute:

```
r = requests.get(f"http://{ip_address}")
r.text
```

You should get the following output:

```
172.28.0.2 - - [03/Nov/2019 04:38:51] "GET / HTTP/1.1" 200 -
'Welcome to my API!'
```

Figure 18.15: Log and result of the GET request to the root endpoint

8. Create a new API endpoint for the **empty** path that accepts only POST requests and will call a function called **check_empty()**. This function will read the JSON received using the **request.get_json()** method, save it into a variable called **data**, and return **True** if **data** is empty or **False** otherwise because JSON using **jsonify()**:

```
@app.route('/empty', methods=['POST'])
def check_empty():
  data = request.get_json()
  return jsonify(not data)
```

9. Create a variable called **empty_json** that will contain an empty JSON by using the **json.dumps()** method with an empty list as a parameter:

```
empty_json = json.dumps([])
```

10. Create a dictionary called headers with the following key-value pairs: **'content-type': 'application/json'**, **'Accept-Charset': 'UTF-8'**:

```
headers = {'content-type': 'application/json', 'Accept-Charset':
'UTF-8'}
```

11. Send a HTTP POST request to the server using the **requests.post()** method with the HTTP URL to the empty endpoint, using **empty_json** and **headers** as its parameters, and print its **.text** attribute:

```
r_empty = requests.post(f"http://{ip_address}/empty", data=empty_
json, headers=headers)
r_empty.text
```

You should get the following output:

```
172.28.0.2 - - [03/Nov/2019 04:38:58] "POST /empty HTTP/1.1" 200 -
'true\n'
```

Figure 18.16: Log and result of a POST request with empty data

12. Create a variable called **not_empty_json** that contains a JSON version of a list, **['Data Science', 'is', 'so', 'cool', '!']**, using the **json.dumps()** method:

```
not_empty_json = json.dumps(['Data Science', 'is', 'so', 'cool',
'!'])
```

13. Send an HTTP POST request to the server using the **`requests.post()`** method with the HTTP URL to the empty endpoint, using **`not_empty_json`** and **`headers`** as its parameters, and print its **`.text`** attribute:

```
r_not_empty = requests.post(f"http://{ip_address}/empty", data=not_
empty_json, headers=headers)
r_not_empty.text
```

You should get the following output:

```
172.28.0.2 - - [03/Nov/2019 04:39:02] "POST /empty HTTP/1.1" 200 -
'false\n'
```

Figure 18.17: Log and result of the POST request with no empty data

Great job! You successfully created a Flask app with two different API endpoints that will display a welcoming message and check whether the input data received is empty or not.

DEPLOYING A MACHINE LEARNING MODEL

Now that we have seen how to build a web API with Flask, we can finally expose our machine learning model via an endpoint. But we need to train a model first. Let's build a classifier with **`RandomForest`** algorithm with the Bank Marketing dataset from *Chapter 3, Binary Classification*.

First, we need to import the required packages:

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

Then we will load the dataset into a DataFrame:

```
file_url = 'https://raw.githubusercontent.com/PacktWorkshops/The-Data-
Science-Workshop/master/Chapter03/bank-full.csv'
df = pd.read_csv(file_url, sep=';')
```

Then we will extract the response variable, which is the **`y`** column in this dataset, using the **`.pop()`** method from **`pandas`**:

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

After this, we need to one-hot encode the categorical variables using the **`.get_dummies()`** method:

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

The final step before modeling is to split the data into training and testing sets. To do so, we will use the **`train_test_split()`** function from **`sklearn`**:

```
X_train, X_test, y_train, y_test = train_test_split(df_dummies, y, test_
size=0.33, random_state=42)
```

Now we can train our RandomForest algorithm.

```
rf_model = RandomForestClassifier(random_state=8)
rf_model.fit(X_train, y_train)
```

We can make predictions on the test set using the **`.predict()`** method:

```
rf_model.predict(X_test)
```

The output will be as follows:

```
array(['no', 'no', 'no', ..., 'no', 'no', 'no'], dtype=object)
```

Figure 18.18: RandomForest predictions on the test set

We can also predict the outcome on a single record from the test set. **`sklearn`** models expect a 2-dimensional array as input, so we need to wrap our record into another list:

```
rf_model.predict([X_test.iloc[3776,]])
```

The output will be as follows:

```
array(['no'], dtype=object)
```

Figure 18.19: RandomForest prediction on record 3776

> **NOTE**
>
> We are not interested in improving the performance of this model. We just need a trained model that we can deploy on our Flask app.

Before adding our model to the Flask app, we need to save it as a file. We will use the **`.dump()`** method from the **`joblib`** package:

```
import joblib
joblib.dump(rf_model, "model.pkl")
```

Your model is saved on the filesystem, and the filename is **`model.pkl`**. To load this model, we can use the **`.load()`** method:

```
saved_model = joblib.load("model.pkl")
```

We can now use it to make predictions:

```
saved_model.predict([X_test.iloc[3776,]])
```

```
array(['no'], dtype=object)
```

Figure 18.20: Prediction of the saved RandomForest model on record 3776

Now we can create a new API endpoint called **`/predict`** that will predict the outcome using this model on the data it receives as input. Within the API function, we need to read the input data, perform the prediction with our pre-loaded model, convert the prediction into a string using the **`array2string`** method from **`numpy`**, and finally convert it to JSON using **`jsonify()`**:

```
import numpy as np

@app.route('/predict', methods=['POST'])
def rf_predict():
  data = request.get_json()
  prediction = saved_model.predict(data)
  str_pred = np.array2string(prediction)
  return jsonify(str_pred)
```

Now we need to send a POST request with the record we want to get prediction from. We will use the same example as previously: record number **`3776`**. First, we need to convert it into a list by using the **`.to_list()`** method from **`pandas`**:

```
record = X_test.iloc[3776,].to_list()
record
```

The output will be as follows:

```
[36,
 229,
 28,
 258,
 2,
 -1,
```

Figure 18.21: Record 3776 converted as list

Then we will transform it into JSON. **`sklearn`** models expect as input a 2-dimensional array, so we need to wrap our input list into another list before calling **`json.dumps()`**:

```
j_data = json.dumps([record])
```

Finally, we can send a POST request with this converted record:

```
r = requests.post("http://172.28.0.2/predict", data=j_data,
headers=headers)
r.text
```

The output will be as follows:

```
172.28.0.2 - - [03/Nov/2019 09:27:40] "POST /predict HTTP/1.1" 200 -
'"[\'no\']"\n'
```

Figure 18.22: Log and prediction of the POST request for the given input record

Great! We got the exact same prediction as before, but this time we got it from our model deployed as a Flask app. As you can see, it is relatively simple to expose a machine learning algorithm as a web API.

EXERCISE 18.02: DEPLOYING A MODEL AS A WEB API

In this exercise, we will learn how to deploy a pre-trained machine learning model as a web API that will accept HTTP POST requests for predicting the class type of breast cancer for a given patient:

> **NOTE**
>
> The dataset used for this exercise is the Breast Cancer Detection dataset shared by Dr. William H. Wolberg from the University of Wisconsin Hospitals, and the attribute information can be found here: https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original).
>
> The dataset can also be found in our repository here: https://packt.live/2QqbHBC.

1. Open a new Colab notebook.

2. Import the **pandas** and **joblib**, **RandomForestClassifier** packages from **sklearn.ensemble** and **train_test_split** from **sklearn.model_selection**:

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

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

```
file_url = 'https://raw.githubusercontent.com/PacktWorkshops/The-Data-Science-Workshop/master/Chapter11/dataset/breast-cancer-wisconsin.data'
```

4. Create a list called **col_names** with the following names: **'Sample code number'**,**'Clump Thickness'**,**'Uniformity of Cell Size'**,**'Uniformity of Cell Shape'**,**'Marginal Adhesion'**,**'Single Epithelial Cell Size'**, **'Bare Nuclei'**,**'Bland Chromatin'**,**'Normal Nucleoli'**,**'Mitoses'**, and **'Class'**:

```
col_names = ['Sample code number','Clump Thickness','Uniformity of Cell Size','Uniformity of Cell Shape','Marginal Adhesion','Single Epithelial Cell Size',
'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
```

5. Load the dataset into a DataFrame using **pd.read_csv()** with the following parameters: **header=None**, **names=col_names**, and **na_values='?'**:

```
df = pd.read_csv(file_url, header=None, names=col_names, na_values='?')
```

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

```
df.head()
```

You should get the following output:

	Sample code number	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape
0	1000025	5	1	1
1	1002945	5	4	4
2	1015425	3	1	1
3	1016277	6	8	8
4	1017023	4	1	1

Figure 18.23: First five rows of the Breast Cancer dataset

7. Replace all missing values with **0** using the **`.fillna()`** method:

```
df.fillna(0, inplace=True)
```

8. Extract the **`'Class'`** response variable using the **`.pop()`** method:

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

9. Remove the **`Sample code number`** column using the **`.drop()`** method with **`axis=1`** as a parameter to specify the fact that we are dropping columns and not rows. Save the result into a DataFrame called **`X`**:

```
X = df.drop('Sample code number', axis=1)
```

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

```
X.head()
```

You should get the following output:

	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape
0	5	1	1
1	5	4	4
2	3	1	1
3	6	8	8
4	4	1	1

Figure 18.24: First five rows of X

11. Split into training and test sets using the **`train_test_split`** function with the parameters **`test_size=0.33`** and **`random_state=888`**:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.33, random_state=888)
```

12. Instantiate a **`RandomForestClassifier`** with **`random_state=1`** and save it into a new variable called **`rf_model`**:

```
rf_model = RandomForestClassifier(random_state=1)
```

13. Train the **`RandomForest`** model with **`X_train`** and **`y_train`**:

```
rf_model.fit(X_train, y_train)
```

You should get the following output:

```
/usr/local/lib/python3.6/dist-packages/sklearn/ensemble/forest.py:245: FutureWarning
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
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=1, verbose=0,
                       warm_start=False)
```

Figure 18.25: Log of the RandomForest model

14. Predict the outcome for the first record of **`X_test`** using the **`.predict()`** method:

```
rf_model.predict([X_test.iloc[0,]])
```

You should get the following output:

```
array([2])
```

Figure 18.26: Prediction of the RandomForest model on the first record of the test set

15. Save the RandomForest model as a separate file called **`model.pkl`** using **`joblib.dump()`**:

```
joblib.dump(rf_model, "model.pkl")
```

You should get the following output:

```
['model.pkl']
```

Figure 18.27: Logs of model saved

16. Import the **socket**, **threading**, **requests**, **json**, and **numpy** packages and the **Flask** class, as well as 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. Save the host IP address into a new variable called **ip_address** using the **.gethostbyname()** and **.gethostname()** methods. Display the value of this new variable:

```
ip_address = socket.gethostbyname(socket.gethostname())
ip_address
```

You should get the following output:

```
'172.28.0.2'
```

Figure 18.28: IP address of the host server

18. Create a **Flask** app and save it into a new variable called **app**:

```
app = Flask(__name__)
```

19. Load the pre-trained model using **joblib.load()**:

```
trained_model = joblib.load("model.pkl")
```

20. 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, predict the outcome with **trained_model**, convert the prediction from a **numpy** array to **string** with **array2string()**, and then convert to JSON with **jsonify()**:

```
@app.route('/api', methods=['POST'])
def predict():
  data = request.get_json()
  prediction = trained_model.predict(data)
  str_pred = np.array2string(prediction)
  return jsonify(str_pred)
```

21. 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
```

Figure 18.29: Log of the Flask app

22. Convert the first record of **`X_test`** into a list and print its content:

```
record = X_test.iloc[0,].to_list()
record
```

You should get the following output:

```
[2.0, 3.0, 1.0, 1.0, 5.0, 1.0, 1.0, 1.0, 1.0]
```

Figure 18.30: Content of first record of test set

23. Create a variable called **`j_data`** that will convert this record into JSON by calling the **`json.dumps()`** method:

```
j_data = json.dumps([record])
```

24. Create a dictionary called headers with the following key-value pairs: **`'content-type': 'application/json'`** and **`'Accept-Charset': 'UTF-8'`**:

```
headers = {'content-type': 'application/json', 'Accept-Charset':
'UTF-8'}
```

25. Send a HTTP POST request to the server using the **`requests.post()`** method with the HTTP URL to the endpoint, using **`j_data`** and **`headers`** as its parameters, and print its **`.text`** attribute:

```
r = requests.post(f"http://{ip_address}/api", data=j_data,
headers=headers)
r.text
```

You should get the following output:

```
172.28.0.2 - - [03/Nov/2019 20:56:13] "POST /api HTTP/1.1" 200 -
'"[2]"\n'
```

Figure 18.31: Logs and prediction from the RandomForest model

You just deployed our pre-trained machine learning algorithm as a web API. In a real-world project, you will have to deploy it on a separate server within your organization, but doing so will not be much more complicated than what you just saw in this exercise.

ADDING DATA PROCESSING LOGIC

In the previous section, we saw that adding an endpoint to a web API for exposing a machine learning algorithm is not much harder than creating any other type of endpoint. There are just a few extra steps required, such as loading the pre-trained model and converting the input and output data into the JSON format. It was quite simple, right?

But the approach we showed you in the previous section has one main drawback: the input data from which we want to get a prediction has been processed beforehand. We performed some data transformation, such as one-hot encoding or handling missing data, on the whole dataset. The record fed to the POST request was already prepared. But in a real project, you will receive new data as input and will need to perform the exact same data preparation steps before predicting its outcome.

This is extremely important as you want the input data to your model to have the same shape and the same type of information as the training set. For example, if you have one-hot encoded a categorical variable that has five different levels on the training set, you will end up with five new binary columns. But if you perform one-hot encoding on a single record, you will get only one new column instead of five. The input data will have four fewer columns and your trained model will throw an error as it expected more columns. What you want is to transform this column into a categorical column with the same list of values as for the training set and then perform one-hot encoding on it. This is exactly what we are going to show you using the Bank Marketing dataset. Let's load it into a DataFrame:

```
import pandas as pd
file_url = 'https://raw.githubusercontent.com/PacktWorkshops/The-Data-
Science-Workshop/master/Chapter03/bank-full.csv'
df = pd.read_csv(file_url, sep=';')
```

Now we are going to extract the response variable called **y**:

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

Before performing any data transformation, we will split the data into training and test sets. But rather than doing a random split, we will use the first **31,647** rows for the training set and the remaining ones for the test set:

```
X_train = df[:31648]
y_train = y[:31648]
X_test = df[31648:]
y_test = y[31648:]
```

Now let's print out the unique values of the **month** column for both training and test sets:

```
print(sorted(X_train['month'].unique()))
print(sorted(X_test['month'].unique()))
```

The output will be as follows:

```
['apr', 'aug', 'dec', 'feb', 'jan', 'jul', 'jun', 'mar', 'may', 'nov', 'oct']
['apr', 'aug', 'dec', 'feb', 'jan', 'jul', 'jun', 'mar', 'may', 'nov', 'oct', 'sep']
```

Figure 18.32: Unique values for the month column in the training and test sets

We can see that there is an additional value in the test set: **sep**. When we perform our prediction, we will need to apply the same data transformation (here, one-hot encoding) as for the training set to any new input data. In this example, what we want is to extract the different category levels (unique values) from each categorical column found in the training set, apply them on new data received, and then perform one-hot encoding. Let's see how we can achieve this in the following steps.

First, let's get the list of all columns of the **object** type:

```
cat_columns = [col for col in X_train.columns if X_train[col].dtype ==
'object']
cat_columns
```

The output will be as follows:

```
['job',
 'marital',
 'education',
 'default',
 'housing',
 'loan',
 'contact',
 'month',
 'poutcome']
```

Figure 18.33: List of columns of the object type

Then we will go through each of these columns, transform them into categorical variables, extract the list of categories using the **`.cat.categories`** attribute from **`pandas`**, and save it into a dictionary:

```
column_categories = {}
for col in cat_columns:
  column_categories[col] = X_train[col].astype('category').cat.categories
```

Let's display the data from this dictionary:

```
column_categories
```

The output will be as follows:

```
{'contact': Index(['cellular', 'telephone', 'unknown'], dtype='object'),
 'default': Index(['no', 'yes'], dtype='object'),
 'education': Index(['primary', 'secondary', 'tertiary', 'unknown'], dtype='object'),
 'housing': Index(['no', 'yes'], dtype='object'),
 'job': Index(['admin.', 'blue-collar', 'entrepreneur', 'housemaid', 'management',
        'retired', 'self-employed', 'services', 'student', 'technician',
        'unemployed', 'unknown'],
       dtype='object'),
 'loan': Index(['no', 'yes'], dtype='object'),
 'marital': Index(['divorced', 'married', 'single'], dtype='object'),
 'month': Index(['apr', 'aug', 'dec', 'feb', 'jan', 'jul', 'jun', 'mar', 'may', 'nov',
        'oct', 'sep'],
       dtype='object'),
 'poutcome': Index(['failure', 'other', 'success', 'unknown'], dtype='object')}
```

Figure 18.34: Dictionary of categories for each column

Now we can use this dictionary to transform the object type columns to categorical with the saved list of categories. To do so, we need to use to the **`CategoricalDtype`** class and provide the list of categories values for the **`categories`** parameter:

```
from pandas.api.types import CategoricalDtype
for col, cat in column_categories.items():
  X_train[col] = X_train[col].astype(CategoricalDtype(categories=cat))
```

Now we can perform one-hot encoding on these categorical columns:

```
X_train_final = pd.get_dummies(X_train, columns=cat_columns)
```

We can train a RandomForest on the prepared training set:

```
from sklearn.ensemble import RandomForestClassifier
rf_model = RandomForestClassifier(random_state=8)
rf_model.fit(X_train_final, y_train)
```

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=8, verbose=0,
                       warm_start=False)
```

Figure 18.35: Log of the trained RandomForest

In order to apply the same data transformation, we have to save this dictionary into a Pickle file (Python file format) using the **`.dump()`** method from the **`pickle`** package:

```
import pickle
pickle.dump(column_categories, open("categories_data.pkl", "wb" ) )
```

Once saved, we can load this file after launching our Flask app and use it to perform transformation on new input data. To load a Pickle file, we need to use the **`.load()`** method from **`pickle`**:

```
categories_data = pickle.load(open("categories_data.pkl", "rb" ) )
categories_data
```

```
{'contact': Index(['cellular', 'telephone', 'unknown'], dtype='object'),
 'default': Index(['no', 'yes'], dtype='object'),
 'education': Index(['primary', 'secondary', 'tertiary', 'unknown'], dtype='object'),
 'housing': Index(['no', 'yes'], dtype='object'),
 'job': Index(['admin.', 'blue-collar', 'entrepreneur', 'housemaid', 'management',
        'retired', 'self-employed', 'services', 'student', 'technician',
        'unemployed', 'unknown'],
       dtype='object'),
 'loan': Index(['no', 'yes'], dtype='object'),
 'marital': Index(['divorced', 'married', 'single'], dtype='object'),
 'month': Index(['apr', 'aug', 'dec', 'feb', 'jan', 'jul', 'jun', 'mar', 'may', 'nov',
        'oct', 'sep'],
       dtype='object'),
 'poutcome': Index(['failure', 'other', 'success', 'unknown'], dtype='object')}
```

Figure 18.36: Loaded dictionary of categories for each column

Now let's try to apply the same data transformation on a record from the test set. We will pick the first record with the **`sep`** value in the month column. We will transform it into JSON using the **`.to_json()`** method from **`pandas`** to simulate the input data that will be received by the API endpoint:

```
record = X_test[X_test['month'] == 'sep'].iloc[0].to_json()
record
```

The output will be as follows:

```
'{"age":23,"job":"student","marital":"single","education":"secondary"
```

Figure 18.37: JSON of the test record

Now we are going to walk you step by step through the code that will be run once the API. First, we will read the JSON from the request:

> **NOTE**
>
> Here we are simulating what the Flask app will do. Instead of using **`request.get_json()`**, we are using **`json.loads()`**.

```
import json
j_data = json.loads(record)
```

When we get the input data, we need to convert it into a DataFrame:

```
df_test = pd.DataFrame(j_data, index=[0])
df_test
```

The output will be as follows:

	age	job	marital	education	default	balance	housing	loan
0	23	student	single	secondary	no	922	no	no

Figure 18.38: DataFrame of a test record

Now we can apply the same conversion to categorical variables as for the training set using the pre-loaded dictionary containing the category levels for each variable:

```
for col, cat in categories_data.items():
  df_test[col] = df_test[col].astype(CategoricalDtype(categories=cat))
```

Then we can perform one-hot encoding on this data:

```
df_test_final = pd.get_dummies(df_test, columns=cat_columns)
```

The output will be as follows:

nth_apr	month_aug	month_dec	month_feb	month_jan	month_jul	month_jun	month_mar	month_may	month_nov	month_oct
0	0	0	0	0	0	0	0	0	0	0

Figure 18.39: One-hot encoded version of the record

After performing one-hot encoding, we can see this record didn't have a new column for **`sep`**. It has the exact same number of columns as for the training set, but as this value is unknown, all the **`month`** columns have the value 0.

Now we can make a prediction using our pre-trained model:

```
rf_model.predict(df_test_final)
```

The output will be as follows:

```
array(['no'], dtype=object)
```

Figure 18.40: RandomForest prediction for this record

In this section, we showed you how to save the main parameters used for preparing the training set and being able to reuse them on new input data. This logic needs to be added into the relevant API endpoint and this what we are going to do in the next exercise.

EXERCISE 18.03: ADDING DATA PROCESSING STEPS INTO A WEB API

In this exercise, we will save the parameters used for processing the training dataset and reuse them on the API we will build to predict the class of cancer for each patient:

> **NOTE**
>
> The dataset used for this exercise is the Breast Cancer dataset shared by Dr. WIlliam H. Wolberg from the University of Wisconsin Hospitals, and the attribute information can be found at https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original).
>
> The dataset can also be found in our repository here: https://packt.live/2QqbHBC.

1. Open a new Colab notebook.
2. Import the **`pandas`** and **`joblib`**, and **`RandomForestClassifier`** packages from **`sklearn.ensemble`**:

```
import pandas as pd
import joblib
from sklearn.ensemble import RandomForestClassifier
```

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

```
file_url = 'https://raw.githubusercontent.com/PacktWorkshops/The-Data-Science-Workshop/master/Chapter11/dataset/breast-cancer-wisconsin.data'
```

4. Create a list called **`col_names`** with the following names: **`'Sample code number'`**,**`'Clump Thickness'`**,**`'Uniformity of Cell Size'`**,**`'Uniformity of Cell Shape'`**,**`'Marginal Adhesion'`**,**`'Single Epithelial Cell Size'`**, **`'Bare Nuclei'`**,**`'Bland Chromatin'`**,**`'Normal Nucleoli'`**,**`'Mitoses'`**, and **`'Class'`**:

```
col_names = ['Sample code number','Clump Thickness','Uniformity of Cell Size','Uniformity of Cell Shape','Marginal Adhesion','Single Epithelial Cell Size',
'Bare Nuclei','Bland Chromatin','Normal Nucleoli','Mitoses','Class']
```

5. Load the dataset into a DataFrame using **`pd.read_csv()`** with the following parameters: **`header=None`**, **`names=col_names`**, and **`na_values='?'`**:

```
df = pd.read_csv(file_url, header=None, names=col_names, na_
values='?')
```

6. Extract the **`'Class'`** response variable using the **`.pop()`** method:

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

7. Remove the **`'Sample code number'`** column from the DataFrame using the **`.drop()`** method with **`axis=1`** as a parameter to specify that we are dropping columns and not rows:

```
df.drop('Sample code number', axis=1, inplace=True)
```

8. Create a variable called **`'training_rows'`** that will contain the number of rows that correspond to **`70%`** of the records:

```
training_rows = int(df.shape[0] * 0.7)
training_rows
```

   You should get the following output:

```
489
```

9. Split the **`df`** and **`y`** DataFrames into training and test sets using **`training_rows`** as the threshold for the split:

```
X_train = df[:training_rows]
y_train = y[:training_rows]
X_test = df[training_rows:]
y_test = y[training_rows:]
```

10. Calculate the number of missing values for each column by combining the **`.isna()`** with **`.sum()`** methods:

```
X_train.isna().sum()
```

You should get the following output:

```
Clump Thickness                0
Uniformity of Cell Size        0
Uniformity of Cell Shape       0
Marginal Adhesion              0
Single Epithelial Cell Size    0
Bare Nuclei                   15
Bland Chromatin                0
Normal Nucleoli                0
Mitoses                        0
dtype: int64
```

Figure 18.41: Number of missing values for each column

11. Extract the list of columns that are not of the **`object`** type and save the result in a variable called **`num_columns`**:

```
num_columns = [col for col in X_train.columns if X_train[col].dtype
!= 'object']
num_columns
```

You should get the following output:

```
['Clump Thickness',
 'Uniformity of Cell Size',
 'Uniformity of Cell Shape',
 'Marginal Adhesion',
 'Single Epithelial Cell Size',
 'Bare Nuclei',
 'Bland Chromatin',
 'Normal Nucleoli',
 'Mitoses']
```

Figure 18.42: List of numerical columns

12. Create an empty dictionary called **`column_mean`**, iterate through the **`num_columns`** list, and for each column, add the column name and its average value to this dictionary and display its content:

```
column_mean = {}
for col in num_columns:
  column_mean[col] = X_train[col].mean()
column_mean
```

You should get the following output:

```
{'Bare Nuclei': 4.0042194092827,
 'Bland Chromatin': 3.61758691206544,
 'Clump Thickness': 4.644171779141105,
 'Marginal Adhesion': 2.9529652351738243,
 'Mitoses': 1.7198364008179958,
 'Normal Nucleoli': 3.1533742331288344,
 'Single Epithelial Cell Size': 3.462167689161554,
 'Uniformity of Cell Shape': 3.4478527607361964,
 'Uniformity of Cell Size': 3.347648261758691}
```

Figure 18.43: Dictionary containing the numerical variables and their average values

13. Import the **`pickle`** package and save **`column_mean`** into a file called **`columns_mean.pkl`**:

```
import pickle
pickle.dump(column_mean, open("columns_mean.pkl", "wb" ) )
```

14. Iterate through the **`num_columns`** list and for each column, replace missing values with the relevant average contained in the **`column_mean`** dictionary:

```
for col in num_columns:
  X_train[col].fillna(column_mean[col], inplace=True)
```

15. Instantiate a **`RandomForestClassifier`** with **`random_state=1`** 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=1)
rf_model.fit(X_train, y_train)
joblib.dump(rf_model, "model.pkl")
```

16. Import the **`socket`**, **`threading`**, **`requests`**, **`json`**, and **`numpy`** packages and the **`Flask`** class, as well as 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 **columns_mean.pkl** using **pickle.load()** and save it into a variable called **var_means**:

```
trained_model = joblib.load("model.pkl")
var_means = pickle.load(open("columns_mean.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, loop through all the items from **var_means**, and use its keys and values to replace missing value, predict the outcome with **trained_model**, convert the prediction from a **numpy** array to a string with **array2string()**, and then convert it to JSON with **jsonify()**:

```
@app.route('/api', methods=['POST'])
def predict():
  data = request.get_json()
  df_test = pd.DataFrame(data, index=[0])
  for col, avg_value in var_means.items():
    df_test[col].fillna(avg_value, inplace=True)
  prediction = trained_model.predict(df_test)
  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** and **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()
```

21. Convert the first record of **X_test** that has missing value on the **'Bare Nuclei'** column and convert it into **json** format using the **.to_json()** method:

```
record = X_test[X_test['Bare Nuclei'].isna()].iloc[0].to_json()
record
```

You should get the following output:

```
'{"Clump Thickness":1.0,"Uniformity of Cell Size":1.0,"Uniformity of Cell Shape":1
```

Figure 18.44: Record converted into JSON format

22. Create a dictionary called headers with the following key-value pairs: **`'content-type'`**: **`'application/json'`** and **`'Accept-Charset'`**: **`'UTF-8'`**. Extract the IP address of the host into a new variable called **`'ip_address'`** 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 03:35:04] "POST /api HTTP/1.1" 200 -
'"[2]"\n'
```

Figure 18.45: Log and prediction of the request received by the API

Congratulations! We have successfully deployed a machine learning model with its data processing artifacts (here, handling missing values) into a web API. Now we are more confident that the input data will have the right shape and type of information expected by the machine learning algorithm. The key here is to save your model and any relevant artifacts into files you will be able to reuse in your web API.

ACTIVITY 18.01: TRAIN AND DEPLOY AN INCOME PREDICTOR MODEL USING FLASK

You are working for a governmental agency and you have been tasked to build and deploy a predictive model using historical census data. The objective of this model is to assess whether a person is more likely to have a salary over or under 50k by looking at their personal information.

The following steps will help you to complete this activity:

1. Download and load the dataset.
2. Extract the response variable.
3. Split the dataset into training and test sets.
4. Extract the list of categories for each categorical column.
5. Save the list of categories and categorical column names into files.
6. Perform one-hot encoding on categorical variables.
7. Train a RandomForest to predict the binary outcome.
8. Save the trained model into a file.
9. Create a Flask app.
10. Create an API endpoint that will perform the same data transformation as for the training set and predict the outcome for a single record.
11. Send a request to this endpoint.

> **NOTE**
>
> The dataset was originally shared by Ron Kohavi, "*Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid*", Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, 1996: https://archive.ics.uci.edu/ml/datasets/Adult
>
> The CSV version of this dataset can be found here: https://www.openml.org/data/get_csv/1595261/phpMawTba
>
> The dataset can be found on our GitHub repository here: https://packt.live/3aBTIAd

The expected output is shown in the following figure:

```
172.28.0.2 - - [06/Nov/2019 11:22:42] "POST /api HTTP/1.1" 200 -
'"[\' <=50K\']"\n'
```

Figure 18.46: Output for deploying an income predictor model using Flask

In this activity, you have trained a machine learning model to assess the likelihood of a person having a low or high salary and you have deployed it to a web API using Flask. This model can now be accessed at any time and can make predictions in real time. You saw how to save the key information required to reproduce the same data processing steps as for the training set. You are now ready to deploy more machine learning models as a service on your own.

SUMMARY

Let's recap what we have learned so far. We saw the basic architecture of a web application, which is composed of two different parts: a client and a server. The most popular protocol used to handle such requests is HTTP. There are multiple types of HTTP requests and we learned how to use the two main ones: GET and POST. From the server, we built multiple endpoints to receive these HTTP requests and processed them in the background using the Flask framework, which is very lightweight but can also easily scale to build more complex web applications. Then we went through the steps required to deploy a pre-trained machine learning model on a web API, and we learned how to use the JSON file format to send and receive data from the server side. Finally, we learned how important it is to not only deploy the model but also the data processing artifacts to ensure the exact same transformation steps used during training will be applied to new incoming prediction requests.

Even though you have learned a lot of new concepts in this chapter, these are just the core basics you need to know to deploy a model. If you want to deploy your model in a real production environment, you may have to consider more advanced topics (that are out of scope of this book), such as adding security layers in order to restrict access to your web API, handling potential errors that may occur, customizing the error messages that will be sent back to the client, and integrating your API with a messaging queue for managing multiple requests.

Now that you have completed this chapter, you have come to the end of this journey through data science. You have studied various topics, ranging from loading a dataset to manipulating it, and from training machine learning algorithms to analyzing and improving their results after exercises and activities. We believe you have the skills required to start your career as a data scientist now, including skills in Python programming, analyzing data, building predictive models, tuning hyperparameters, analyzing model performance, and deploying models to production.

We hope you enjoyed learning all the concepts presented in this book and are now eager to start a data science project on your own and learn more interesting and advanced techniques.