CHAPTER 9

Deploying Machine Learning Models

This chapter will educate you on best practices to follow when deploying a machine learning model. We will use three methods to demonstrate the production deployment, as follows:

- Model deployment using HDFS object and pickle files
- Model deployment using Docker
- Real-time scoring API

In the previous chapter, you got a glimpse of the first technique. In this section, we will cover it in more detail. The code provided is the starter code to build the model and evaluate the results.

Starter Code

This code is just a template script for you to load the *bank-full.csv* file and build a model. If you already have a model, you can skip this section.

```
#default parameters
filename = "bank-full.csv"
target_variable_name = "y"

#load datasets
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
df = spark.read.csv(filename, header=True, inferSchema=True, sep=';')
df.show()
```

```
#convert target column variable to numeric
from pyspark.sql import functions as F
df = df.withColumn('y', F.when(F.col("y") == 'yes', 1).otherwise(0))

#datatypes of each column and treat them accordingly
# identify variable types function
def variable_type(df):
    vars_list = df.dtypes
    char_vars = []
    num_vars = []
    for i in vars_list:
        if i[1] in ('string'):
            char_vars.append(i[0])
        else:
            num_vars.append(i[0])
    return char_vars, num_vars

char_vars, num_vars = variable_type(df)
num_vars.remove(target_variable_name)

from pyspark.ml.feature import StringIndexer
from pyspark.ml import Pipeline

# convert categorical to numeric using label encoder option
def category_to_index(df, char_vars):

    char_df = df.select(char_vars)
    indexers = [StringIndexer(inputCol=c, outputCol=c+"_index",
    handleInvalid="keep") for c in char_df.columns]
    pipeline = Pipeline(stages=indexers)
    char_labels = pipeline.fit(char_df)
    df = char_labels.transform(df)
    return df, char_labels

df, char_labels = category_to_index(df, char_vars)
df = df.select([c for c in df.columns if c not in char_vars])

# rename encoded columns to original variable name
def rename_columns(df, char_vars):
    mapping = dict(zip([i + '_index' for i in char_vars], char_vars))
    df = df.select([F.col(c).alias(mapping.get(c, c)) for c in df.columns])
    return df

df = rename_columns(df, char_vars)

from pyspark.ml.feature import VectorAssembler

#assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name):
    stages = []
    #assemble vectors
    assembler = VectorAssembler(inputCols=features_list,
    outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [target_variable_name, 'features'] + features_list
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
    #assembler model
    assembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = assembleModel.transform(df).select(selectedCols)

    return df, assembleModel, selectedCols

#exclude target variable and select all other feature vectors
features_list = df.columns
#features_list = char_vars #this option is used only for ChiSqselector
features_list.remove(target_variable_name)

# apply the function on our DataFrame
df, assembleModel, selectedCols = assemble_vectors(df, features_list,
target_variable_name)

#train test split - 70/30 split
train, test = df.randomSplit([0.7, 0.3], seed=12345)
```

```
train.count(), test.count()

# Random forest model
from pyspark.ml.classification import RandomForestClassifier

clf = RandomForestClassifier(featuresCol='features', labelCol='y')
clf_model = clf.fit(train)
print(clf_model.featureImportances)
print(clf_model.toDebugString)
train_pred_result = clf_model.transform(train)
test_pred_result = clf_model.transform(test)

# Validate random forest model
from pyspark.mllib.evaluation import MulticlassMetrics
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.sql.types import IntegerType, DoubleType

def evaluation_metrics(df, target_variable_name):
    pred = df.select("prediction", target_variable_name)
    pred = pred.withColumn(target_variable_name, pred[target_variable_
           name].cast(DoubleType()))
    pred = pred.withColumn("prediction", pred["prediction"].
           cast(DoubleType()))
    metrics = MulticlassMetrics(pred.rdd.map(tuple))
    # confusion matrix
    cm = metrics.confusionMatrix().toArray()
    acc = metrics.accuracy #accuracy
    misclassification_rate = 1 - acc #misclassification rate
    precision = metrics.precision(1.0) #precision
    recall = metrics.recall(1.0) #recall
    f1 = metrics.fMeasure(1.0) #f1-score
    #roc value
    evaluator_roc = BinaryClassificationEvaluator
                    (labelCol=target_variable_name, rawPredictionCol='rawPr
                    ediction', metricName='areaUnderROC')
    roc = evaluator_roc.evaluate(df)
    evaluator_pr = BinaryClassificationEvaluator
                   (labelCol=target_variable_name, rawPredictionCol='rawPre
                   diction', metricName='areaUnderPR')
    pr = evaluator_pr.evaluate(df)
    return cm, acc, misclassification_rate, precision, recall, f1, roc, pr
    train_cm, train_acc, train_miss_rate, train_precision, \
        train_recall, train_f1, train_roc, train_pr = evaluation_
        metrics(train_pred_result, target_variable_name)
test_cm, test_acc, test_miss_rate, test_precision, \
        test_recall, test_f1, test_roc, test_pr = evaluation_metrics(test_
        pred_result, target_variable_name)

#Comparision of metrics
print('Train accuracy - ', train_acc, ', Test accuracy - ', test_acc)
print('Train misclassification rate - ', train_miss_rate, ', Test
misclassification rate - ', test_miss_rate)
print('Train precision - ', train_precision, ', Test precision - ', test_
precision)
print('Train recall - ', train_recall, ', Test recall - ', test_recall)
print('Train f1 score - ', train_f1, ', Test f1 score - ', test_f1)
print('Train ROC - ', train_roc, ', Test ROC - ', test_roc)
print('Train PR - ', train_pr, ', Test PR - ', test_pr)

# make confusion matrix charts
import seaborn as sns
import matplotlib.pyplot as plt

def make_confusion_matrix_chart(cf_matrix_train, cf_matrix_test):

    list_values = ['0', '1']

    plt.figure(1, figsize=(10,5))
    plt.subplot(121)
    sns.heatmap(cf_matrix_train, annot=True, yticklabels=list_values,
                                 xticklabels=list_values, fmt='g')
    plt.ylabel("Actual")
    plt.xlabel("Pred")
    plt.ylim([0,len(list_values)])
    plt.title('Train data predictions')

    plt.subplot(122)
    sns.heatmap(cf_matrix_test, annot=True, yticklabels=list_values,
                                xticklabels=list_values, fmt='g')
    plt.ylabel("Actual")
    plt.xlabel("Pred")
    plt.ylim([0,len(list_values)])
    plt.title('Test data predictions')

    plt.tight_layout()
    return None
make_confusion_matrix_chart(train_cm, test_cm)

#Make ROC chart and PR curve
from pyspark.mllib.evaluation import BinaryClassificationMetrics

class CurveMetrics(BinaryClassificationMetrics):
    def __init__(self, *args):
        super(CurveMetrics, self).__init__(*args)

    def _to_list(self, rdd):
        points = []
        results_collect = rdd.collect()
        for row in results_collect:
            points += [(float(row._1()), float(row._2()))]
        return points

    def get_curve(self, method):
        rdd = getattr(self._java_model, method)().toJavaRDD()
        return self._to_list(rdd)

import matplotlib.pyplot as plt


def plot_roc_pr(df, target_variable_name, plot_type, legend_value, title):


    preds = df.select(target_variable_name,'probability')
    preds = preds.rdd.map(lambda row: (float(row['probability'][1]),
    float(row[target_variable_name])))
    # Returns as a list (false positive rate, true positive rate)
    points = CurveMetrics(preds).get_curve(plot_type)
    plt.figure()
    x_val = [x[0] for x in points]
    y_val = [x[1] for x in points]
    plt.title(title)

    if plot_type == 'roc':
        plt.xlabel('False Positive Rate (1-Specificity)')
        plt.ylabel('True Positive Rate (Sensitivity)')
        plt.plot(x_val, y_val, label = 'AUC = %0.2f' % legend_value)
        plt.plot([0, 1], [0, 1], color='red', linestyle='--')

    if plot_type == 'pr':
        plt.xlabel('Recall')
        plt.ylabel('Precision')
        plt.plot(x_val, y_val, label = 'Average Precision = %0.2f' %
        legend_value)
        plt.plot([0, 1], [0.5, 0.5], color='red', linestyle='--')

    plt.legend(loc = 'lower right')
    return None

plot_roc_pr(train_pred_result, target_variable_name, 'roc', train_roc,
'Train ROC')
plot_roc_pr(test_pred_result, target_variable_name, 'roc', test_roc, 'Test
ROC')
plot_roc_pr(train_pred_result, target_variable_name, 'pr', train_pr, 'Train
Precision-Recall curve')
plot_roc_pr(test_pred_result, target_variable_name, 'pr', test_pr, 'Test
Precision-Recall curve')
```

Save Model Objects and Create Score Code

So far, we have built a solid model and validated it using multiple metrics. It's time to save the model objects for future scoring and create score code.

- Model objects are the PySpark or Python parameters that contain information from the training process. We have to identify all the relevant model objects that would be needed to replicate the results without retraining the model. Once identified, they are saved in a specific location (HDFS or other path) and used later during scoring.
- The score code contains the essential code to run your model objects on the new dataset so as to produce the scores. Remember, the score code is used only for the scoring process. You should not have the model-training scripts in a score code.

Model Objects

In this model example, we need six objects to be saved for future use, as follows:

- `char_labels` – This object is used to apply label encoding on new data.
- `assembleModel` – This object is used to assemble vectors and make the data ready for scoring.
- `clf_model` – This object is the random forest classifier model.
- `features_list` – This object has the list of input features to be used from the new data.
- `char_vars` – character variables
- `num_vars` – numeric variables

Below code saves all the objects and pickle files required for scoring in the path provided.

```
import os
import pickle

path_to_write_output = '/home/work/Desktop/score_code_objects'
#create directoyr, skip if already exists
try:
    os.mkdir(path_to_write_output)
except:
    pass

#save pyspark objects
char_labels.write().overwrite().save(path_to_write_output + '/char_label_
model.h5')
assembleModel.write().overwrite().save(path_to_write_output + '/
assembleModel.h5')
clf_model.write().overwrite().save(path_to_write_output + '/clf_model.h5')

#save python object
list_of_vars = [features_list, char_vars, num_vars]
with open(path_to_write_output + '/file.pkl', 'wb') as handle:
    pickle.dump(list_of_vars, handle)
```

Score Code

This is a Python script file used to perform scoring operations. As mentioned before, this script file should not contain model-training scripts. At bare minimum, a score code should do the following tasks:

1) Import necessary packages
2) Read the new input file to perform scoring
3) Read the model objects saved from the training process
4) Perform necessary transformations using model objects
5) Calculate model scores using the classifier/regression object
6) Output the final scores in a specified location

First, we will create a *helper.py* script that has all the necessary code to perform scoring. The code to paste in the file is provided here:

```
#helper functions - helper.py file
from pyspark.sql import functions as F
import pickle
from pyspark.ml import PipelineModel
from pyspark.ml.classification import RandomForestClassificationModel
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType, DoubleType

# read model objects saved from the training process
path_to_read_objects = '/home/work/Desktop/score_code_objects'

#pyspark objects
char_labels = PipelineModel.load(path_to_read_objects + '/char_label_model.
h5')
assembleModel = PipelineModel.load(path_to_read_objects + '/assembleModel.
h5')
clf_model = RandomForestClassificationModel.load(path_to_read_objects + '/
clf_model.h5')
#python objects
with open(path_to_read_objects + '/file.pkl', 'rb') as handle:
    features_list, char_vars, num_vars = pickle.load(handle)

#make necessary transformations
def rename_columns(df, char_vars):
    mapping = dict(zip([i + '_index' for i in char_vars], char_vars))
    df = df.select([F.col(c).alias(mapping.get(c, c)) for c in df.columns])
    return df

# score the new data
def score_new_df(scoredf):
    X = scoredf.select(features_list)
    X = char_labels.transform(X)
    X = X.select([c for c in X.columns if c not in char_vars])
    X = rename_columns(X, char_vars)
    final_X = assembleModel.transform(X)
    final_X.cache()
    pred = clf_model.transform(final_X)
    pred.cache()
    split_udf = udf(lambda value: value[1].item(), DoubleType())
    pred = pred.select('prediction', split_udf('probability').
    alias('probability'))
    return pred
```

After you create the file, you can score new data using the following code. You can save the scripts in a *run.py* file.

```
# import necessary packages
from pyspark.sql import SparkSession
from helper import *

# new data to score
path_to_output_scores = '.'
filename = "score_data.csv"
spark = SparkSession.builder.getOrCreate()
score_data = spark.read.csv(filename, header=True, inferSchema=True,
sep=';')

# score the data
final_scores_df = score_new_df(score_data)
#final_scores_df.show()
final_scores_df.repartition(1).write.format('csv').mode("overwrite").
options(sep='|', header='true').save(path_to_output_scores + "/predictions.
csv")
```

The final scores are available in the *final_scores_df* DataFrame, and this dataset is saved as a csv file in the final step. You need to make sure that the *run.py* and *helper.py* files exist in the same directory. That's it. We have a solid score code that is ready to be deployed in a production environment. We will go through the production implementation in the next sections.

Model Deployment Using HDFS Object and Pickle Files

This is a very simple way to deploy a model in production. When you have multiple models to score, you can manually submit each *run.py* script that you created in the previous step, or create a scheduler to run the scripts periodically. In either case, the following code will help you accomplish this task:

```
spark-submit run.py
```

As mentioned before, the *run.py* and *helper.py* files should be in the same directory for this code to work. You can configure the preceding command to specify `configurations` while submitting the job. The `configurations` are useful when you deal with large datasets, because you might have to play with executors, cores, and memory options. You can read about the `spark-submit` and `configuration` options at the following links:

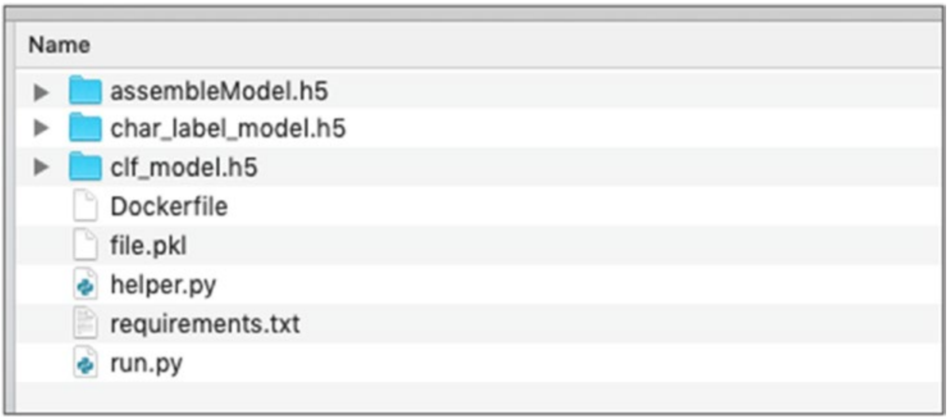
https://spark.apache.org/docs/latest/submitting-applications.html
https://spark.apache.org/docs/latest/configuration.html

Model Deployment Using Docker

Let's switch gears and deploy a model using Docker. Docker-based deployment is useful in a lot of ways when compared to a traditional scoring method, like `spark-submit`.

- Portability – You can port your application to any platform and instantly make it work.
- Containers – The entire application is self-contained. All the codes and executables needed to make your application work sit within the container.
- Microservices architecture – Instead of a traditional, monolithic application, we can separate it into micro-services. This way, we can modify or scale up/down individual micro-services as and when needed.
- Faster software delivery cycles – Compatible with Continuous Integration and Continuous Development (CI/CD) processes so that application can be easily deployed to production
- Data scientists/engineers don't have to spend additional time to fix their codes/bugs when the Spark backend is altered.
- When compared to virtual machines, Docker consumes less memory, thus making it more efficient in using system resources.

Let's go ahead and implement our model using Docker. Before we start, we should have the following files in a single directory (Figure 9-1).



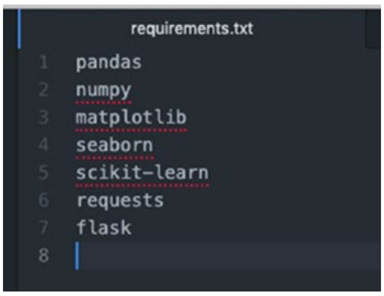
```
Name
▶ assembleModel.h5
▶ char_label_model.h5
▶ clf_model.h5
  Dockerfile
  file.pkl
  helper.py
  requirements.txt
  run.py
```

Figure 9-1. *Directory structure*

Most of the files should look familiar to you by now. We have two new files shown in the image: *Dockerfile* and *requirements.txt*. The *Dockerfile* contains the scripts to create a Docker container, and the *requirements.txt* file contains all the additional packages that you need to make your code work. Let's look at the file contents.

requirements.txt file

The contents of this file are provided here so that you can copy and paste them inside the file (Figure 9-2).

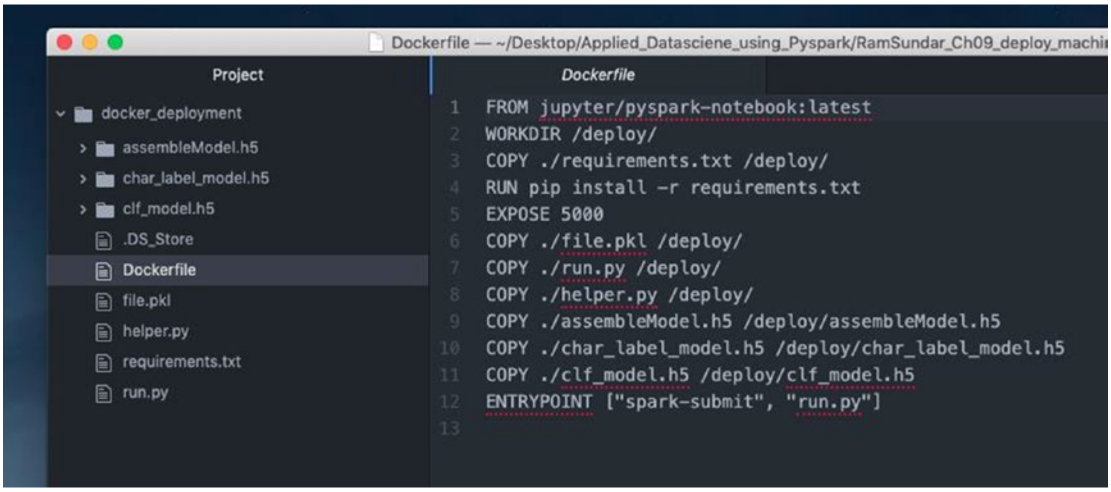


```
requirements.txt
pandas
numpy
matplotlib
seaborn
scikit-learn
requests
flask
```

Figure 9-2. *requirements.txt file*

Dockerfile

We are using the base Docker image as the PySpark Jupyter Notebook image. Inside the image, we create a working directory called `/deploy/` into which we will copy and paste all our model objects and files (Figure 9-3).


Dockerfile — ~/Desktop/Applied_Datasciene_using_Pyspark/RamSundar_Ch09_deploy_machi

Project

- docker_deployment
  - assembleModel.h5
  - char_label_model.h5
  - clf_model.h5
  - .DS_Store
  - Dockerfile
  - file.pkl
  - helper.py
  - requirements.txt
  - run.py

Dockerfile

```
FROM jupyter/pyspark-notebook:latest
WORKDIR /deploy/
COPY ./requirements.txt /deploy/
RUN pip install -r requirements.txt
EXPOSE 5000
COPY ./file.pkl /deploy/
COPY ./run.py /deploy/
COPY ./helper.py /deploy/
COPY ./assembleModel.h5 /deploy/assembleModel.h5
COPY ./char_label_model.h5 /deploy/char_label_model.h5
COPY ./clf_model.h5 /deploy/clf_model.h5
ENTRYPOINT ["spark-submit", "run.py"]
```

Figure 9-3. *Dockerfile*

This is done in successive steps. In addition, we install the required packages by executing the *requirements.txt* file, which is shown in line 4. Finally, we execute the model using the `spark-submit` command used in the previous section. One thing to notice is that the port number, 5000, is exposed in the container. We will use this port in the real-time scoring section. Just ignore this port for now. The *Dockerfile* script is provided here:

```
FROM jupyter/pyspark-notebook:latest
WORKDIR /deploy/
COPY ./requirements.txt /deploy/
RUN pip install -r requirements.txt
EXPOSE 5000
COPY ./file.pkl /deploy/
COPY ./run.py /deploy/
COPY ./helper.py /deploy/
COPY ./assembleModel.h5 /deploy/assembleModel.h5
COPY ./char_label_model.h5 /deploy/char_label_model.h5
COPY ./clf_model.h5 /deploy/clf_model.h5
ENTRYPOINT ["spark-submit", "run.py"]
```

Changes Made in helper.py and run.py Files

In the *helper.py* file, change the following line:

```
path_to_read_objects ='/deploy'
```

In the *run.py* file, change the following lines:

```
path_to_output_scores = '/localuser'
filename = path_to_output_scores + "/score_data.csv"
```

These changes are made to reflect the appropriate directory path within Docker (`/deploy`) and the volume that we will mount later (`/localuser`) during Docker execution.

Create Docker and Execute Score Code

To create the Docker container, execute the following code in the Terminal/command prompt:

```
docker build -t scoring_image .
```

Once the image is built, you can execute the image using the following command. You need to execute the command from the directory where the scoring data exists.

```
docker run -p 5000:5000 -v ${PWD}:/localuser scoring_image:latest
```

That's it. You should see the *predictions.csv* file in the local folder. Until now, we have used a batch-scoring process. Let's now move toward a real-time scoring API.

Real-Time Scoring API

A real-time scoring API is used to score your models in real-time. It enables your model to be embedded in a web browser so as to get predictions and take action based on predictions. This is useful for making online recommendations, checking credit card approval status, and so on. The application is enormous, and it becomes easier to manage such APIs using Docker. Let's use Flask to implement our model in real-time (Figure 9-4). The flask scripts are stored in the *app.py* file, the contents of which are provided next.

app.py File

```
#app.py file
from flask import Flask, request, redirect, url_for, flash, jsonify, make_
response
import numpy as np
import pickle
import json
import os, sys
# Path for spark source folder
os.environ['SPARK_HOME'] = '/usr/local/spark'
# Append pyspark  to Python Path
sys.path.append('/usr/local/spark/python')

from pyspark.sql import SparkSession
from pyspark import SparkConf, SparkContext
from helper import *

conf = SparkConf().setAppName("real_time_scoring_api")
conf.set('spark.sql.warehouse.dir', 'file:///usr/local/spark/spark-
warehouse/')
conf.set("spark.driver.allowMultipleContexts", "true")
spark = SparkSession.builder.master('local').config(conf=conf).
getOrCreate()
sc = spark.sparkContext

app = Flask(__name__)
```

```
@app.route('/api/', methods=['POST'])
def makecalc():

    json_data = request.get_json()
    #read the real time input to pyspark df
    score_data = spark.read.json(sc.parallelize(json_data))
    #score df
    final_scores_df = score_new_df(score_data)
    #convert predictions to Pandas DataFrame
    pred = final_scores_df.toPandas()
    final_pred = pred.to_dict(orient='rows')[0]
    return jsonify(final_pred)

if __name__ == '__main__':

    app.run(debug=True, host='0.0.0.0', port=5000)
```

We will also change our *Dockerfile* to copy the *app.py* file to the `/deploy` directory. In the last module, we ran the *run.py* file. In this module, it is taken care of by the *app.py* file. Here is the updated *Dockerfile* content:

```
FROM jupyter/pyspark-notebook:latest
WORKDIR /deploy/
COPY ./requirements.txt /deploy/
RUN pip install -r requirements.txt
EXPOSE 5000
COPY ./file.pkl /deploy/
COPY ./run.py /deploy/
COPY ./helper.py /deploy/
COPY ./assembleModel.h5 /deploy/assembleModel.h5
COPY ./char_label_model.h5 /deploy/char_label_model.h5
COPY ./clf_model.h5 /deploy/clf_model.h5
COPY ./app.py /deploy/
ENTRYPOINT ["python", "app.py"]
```

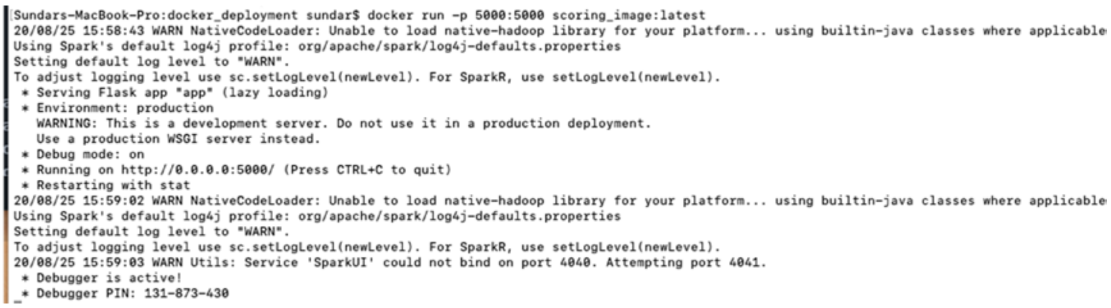
Okay, we are set to perform real-time scoring. We will code the UI shortly. Let's first test the code using the `Postman` application.

Recreate the Docker image using the following code:

```
docker build -t scoring_image .
```

You won't require volume mapping this time, since we are going to perform live scoring. You need to make sure that the port number of the local system is mapped to the Docker port number.

```
docker run -p 5000:5000 scoring_image:latest
```



```
Sundars-MacBook-Pro:docker_deployment sundar$ docker run -p 5000:5000 scoring_image:latest
20/08/25 15:58:43 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
 * Serving Flask app "app" (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: on
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 * Restarting with stat
20/08/25 15:59:02 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
20/08/25 15:59:03 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
 * Debugger is active!
 * Debugger PIN: 131-873-430
```

Figure 9-4. *Flask API ready*

Once you get this message, you can switch back to the Postman API and test this app. You can download Postman online and install it if you don't have it already. Initiate the application and do the necessary changes, as shown next.

Postman API

This section is intended for readers who don't have a background in the Postman API. Please skip this section if you are already familiar with it (Figure 9-5).

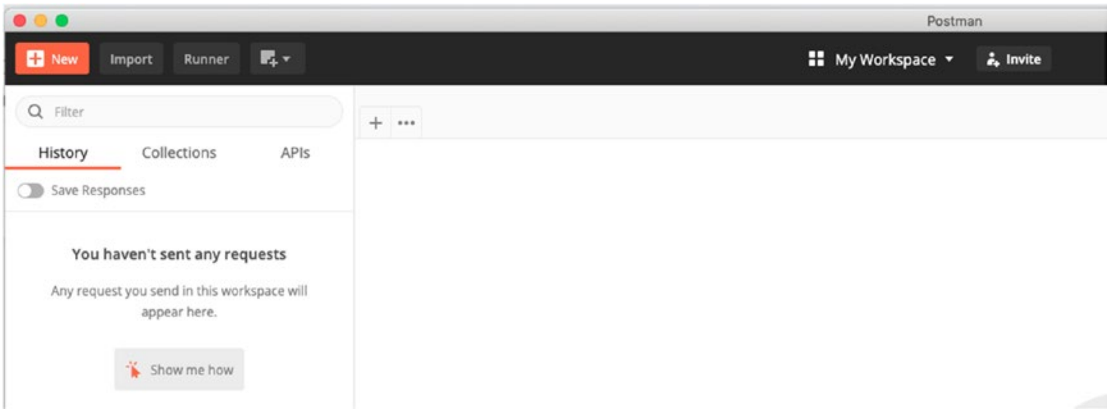
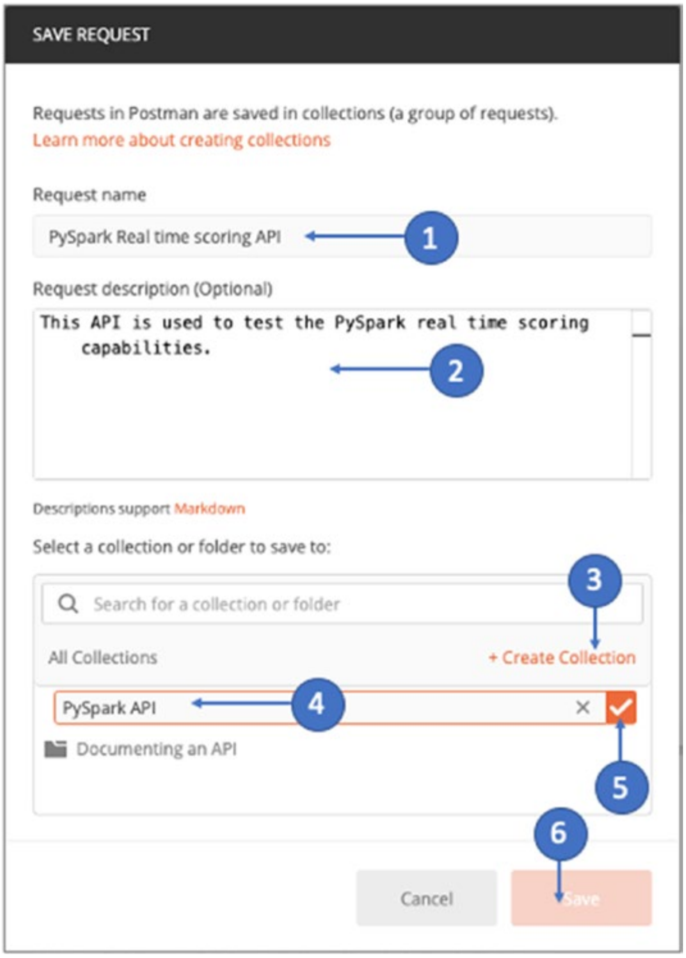


Figure 9-5. *Postman API*

You need to create an API instance by clicking the *New* button in the left corner and then clicking on *Request Create a basic request* to get the window shown in Figure 9-6.


SAVE REQUEST

Requests in Postman are saved in collections (a group of requests).
Learn more about creating collections

Request name
PySpark Real time scoring API

Request description (Optional)
This API is used to test the PySpark real time scoring capabilities.

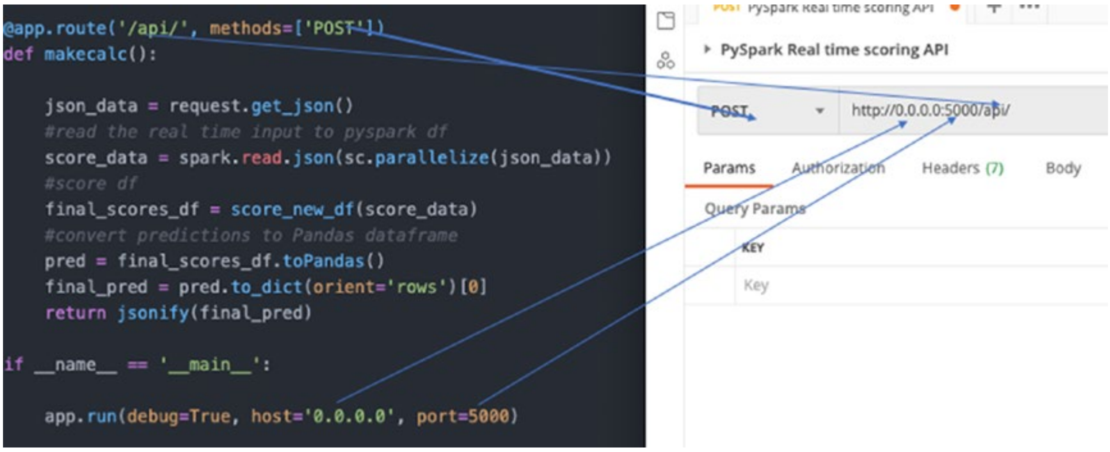
Descriptions support Markdown

Select a collection or folder to save to:
Search for a collection or folder
All Collections
+ Create Collection
PySpark API
Documenting an API

Cancel
Save

Figure 9-6. *Create a new API request.*

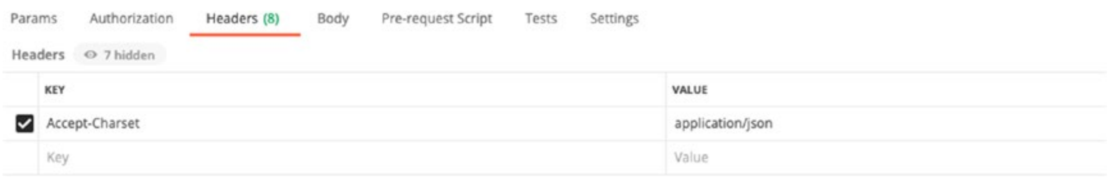
Now, let's make the configurations in the API shown in Figure 9-7.



```
@app.route('/api/', methods=['POST'])
def makecalc():

    json_data = request.get_json()
    #read the real time input to pyspark df
    score_data = spark.read.json(sc.parallelize(json_data))
    #score df
    final_scores_df = score_new_df(score_data)
    #convert predictions to Pandas dataframe
    pred = final_scores_df.toPandas()
    final_pred = pred.to_dict(orient='rows')[0]
    return jsonify(final_pred)

if __name__ == '__main__':

    app.run(debug=True, host='0.0.0.0', port=5000)
```

Figure 9-7. Configurations

Let's add a header column by navigating to the *Headers* tab. With this option, the API knows that it is looking for an JSON input (Figure 9-8).



	KEY	VALUE
☑	Accept-Charset	application/json
	Key	Value

Figure 9-8. JSON header object

Finally, you are ready to score. Since you are providing the input for testing, let's switch the body type to `raw` and select `JSON` as the text option (Figure 9-9).

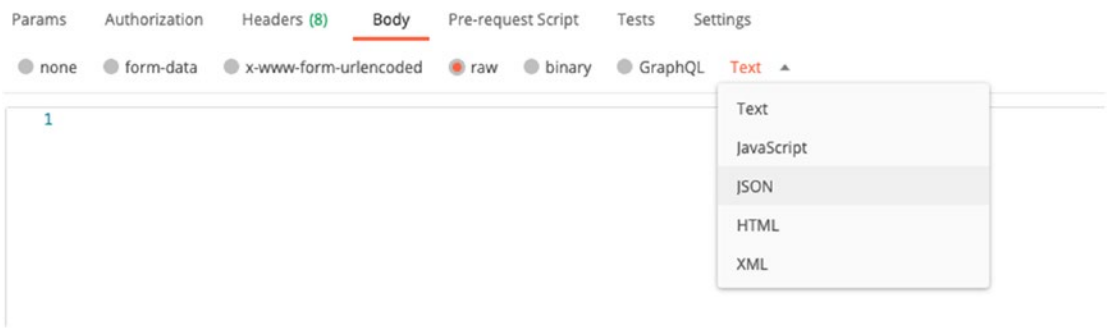


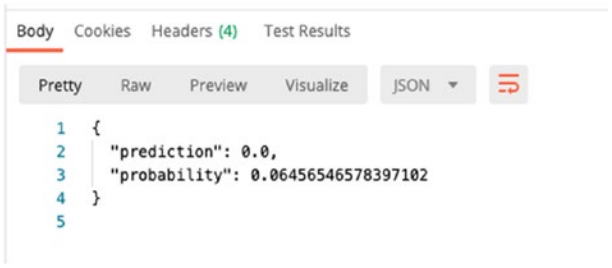
Figure 9-9. JSON selection

Test Real-Time Using Postman API

Use the following JSON object for testing purposes. Note the square brackets in front. This is used to pass a list of JSON objects.

```
[{"age":58,"job":"management","marital":"married","education":"tertiary","d
efault":"no","balance":2143,"housing":"yes","loan":"no","contact":"unknown"
,"day":5,"month":"may","duration":261,"campaign":1,"pdays":-1,"previous":0,
"poutcome":"unknown"}]
```

Copy and paste the JSON object into the body of the API and click *Send* (Figure 9-10).



```
{
  "prediction": 0.0,
  "probability": 0.06456546578397102
}
```

Figure 9-10. *Real-time scores*

We can now roc proceed to build the UI component for our app.

Build UI

We will use a Python package `streamlit` to build the user interface (UI). `streamlit` is easy to integrate with our existing code. One of the greatest advantages of using `streamlit` is that you can edit the Python code and see live changes in the web app instantly. Another advantage is that it requires minimal code to build the UI. Before we get started, let's look at the directory structure shown in Figure 9-11. The parent directory is `real_time_scoring`. We have created two child directories: `pysparkapi` and `streamlitapi`. We will move all our code discussed so far to the `pysparkapi` directory. The UI code will reside in the `streamlitapi` directory.

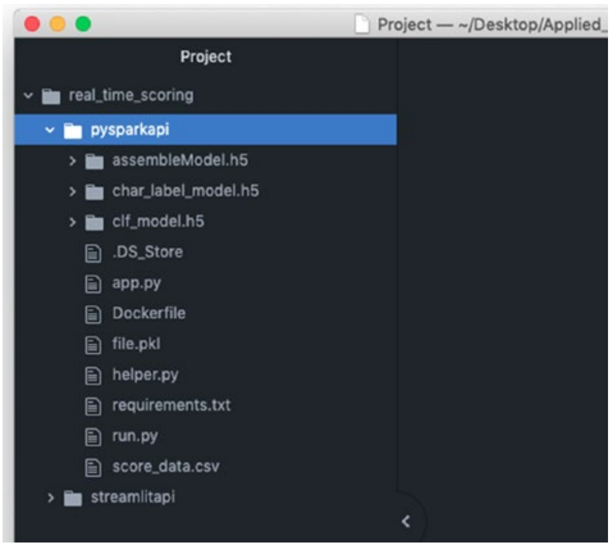


Figure 9-11. *UI directory structure*

The streamlitapi Directory

Let's navigate to the `streamlitapi` directory and create the following three files within it.

Dockerfile

```
FROM python:3.7-slim
WORKDIR /streamlit
COPY ./requirements.txt /streamlit
RUN pip install -r requirements.txt
COPY ./webapp.py /streamlit
EXPOSE 8501
CMD ["streamlit", "run", "webapp.py"]
```

requirements.txt

```
streamlit
requests
```

The following file contains the UI code.

webapp.py

```
import streamlit as st
import requests
import datetime
import json

st.title('PySpark Real Time Scoring API')

# PySpark API endpoint
url = 'http://pysparkapi:5000'
endpoint = '/api/'

# description and instructions
st.write('''A real time scoring API for PySpark model.''')

st.header('User Input features')

def user_input_features():
    input_features = {}
    input_features["age"] = st.slider('Age', 18, 95)
    input_features["job"] = st.selectbox('Job', ['management',
    'technician', 'entrepreneur', 'blue-collar', \
     'unknown', 'retired', 'admin.', 'services', 'self-employed', \
           'unemployed', 'housemaid', 'student'])
    input_features["marital"] = st.selectbox('Marital Status', ['married',
    'single', 'divorced'])
    input_features["education"] = st.selectbox('Education Qualification',
    ['tertiary', 'secondary', 'unknown', 'primary'])
    input_features["default"] = st.selectbox('Have you defaulted before?',
    ['yes', 'no'])
    input_features["balance"]= st.slider('Current balance', -10000, 150000)
    input_features["housing"] = st.selectbox('Do you own a home?', ['yes',
    'no'])
    input_features["loan"] = st.selectbox('Do you have a loan?', ['yes',
    'no'])
    input_features["contact"] = st.selectbox('Best way to contact you',
    ['cellular', 'telephone', 'unknown'])
    date = st.date_input("Today's Date")
    input_features["day"] = date.day
    input_features["month"] = date.strftime("%b").lower()
    input_features["duration"] = st.slider('Duration', 0, 5000)
    input_features["campaign"] = st.slider('Campaign', 1, 63)
    input_features["pdays"] = st.slider('pdays', -1, 871)
    input_features["previous"] = st.slider('previous', 0, 275)
    input_features["poutcome"] = st.selectbox('poutcome', ['success',
    'failure', 'other', 'unknown'])
    return [input_features]

json_data = user_input_features()

submit = st.button('Get predictions')
if submit:
    results = requests.post(url+endpoint, json=json_data)
    results = json.loads(results.text)
    st.header('Final Result')
    prediction = results["prediction"]
    probability = results["probability"]
    st.write("Prediction: ", int(prediction))
    st.write("Probability: ", round(probability,3))
```

You can test this code before deploying in a Docker container by running the following code:

```
streamlit run webapp.py
```

As mentioned before, the interactive feature of `streamlit` lets you design custom layouts on the fly with minimal code. Okay, we are all set. These three files reside in the `streamlitapi` directory. We now have two Docker containers—one for PySpark flask and the second one for the `streamlit` UI. We need to create a network between these two containers so that the UI interacts with the backend PySpark API. Let's switch to the parent directory `real_time_scoring`.

real_time_scoring Directory

A *docker-compose* file is used to combine multiple Docker containers and create a Docker network service. Let's look at the contents of the file.

docker-compose.yml

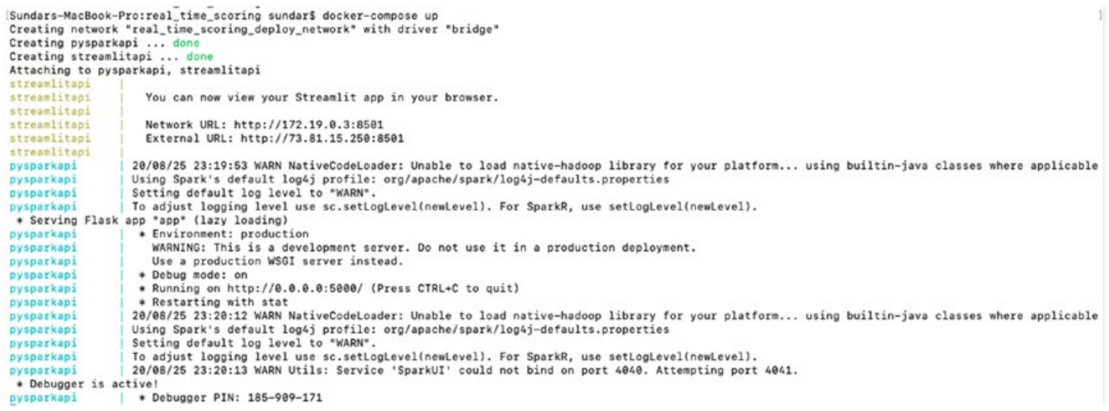
```
version: '3'

services:
  pysparkapi:
    build: pysparkapi/
    ports:
      - 5000:5000
    networks:
      - deploy_network
    container_name: pysparkapi

  streamlitapi:
    build: streamlitapi/
    depends_on:
      - pysparkapi
    ports:
        - 8501:8501
    networks:
      - deploy_network
    container_name: streamlitapi

networks:
  deploy_network:
    driver: bridge
```

Executing docker-compose.yml File

Let's use the following code to execute the `docker-compose.yml` file:

```
docker-compose build
```

The Docker network service is established. Let's execute the Docker service using the code shown in Figure 9-12. It will take time to spin up the service and wait for the debugger pin to appear before you switch to your browser.



```
Sundars-MacBook-Pro:real_time_scoring sundar$ docker-compose up
Creating network "real_time_scoring_deploy_network" with driver "bridge"
Creating pysparkapi ... done
Creating streamlitapi ... done
Attaching to pysparkapi, streamlitapi
streamlitapi    |
streamlitapi    |   You can now view your Streamlit app in your browser.
streamlitapi    |
streamlitapi    |   Network URL: http://172.19.0.3:8501
streamlitapi    |   External URL: http://73.81.15.250:8501
streamlitapi    |
pysparkapi      | 20/08/25 23:19:53 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
pysparkapi      | Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
pysparkapi      | Setting default log level to "WARN".
pysparkapi      | To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
 * Serving Flask app "app" (lazy loading)
pysparkapi      |  * Environment: production
pysparkapi      |    WARNING: This is a development server. Do not use it in a production deployment.
pysparkapi      |    Use a production WSGI server instead.
pysparkapi      |  * Debug mode: on
pysparkapi      |  * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
pysparkapi      |  * Restarting with stat
pysparkapi      | 20/08/25 23:20:12 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
pysparkapi      | Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
pysparkapi      | Setting default log level to "WARN".
pysparkapi      | To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
pysparkapi      | 20/08/25 23:20:13 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
 * Debugger is active!
pysparkapi      |  * Debugger PIN: 185-909-171
```

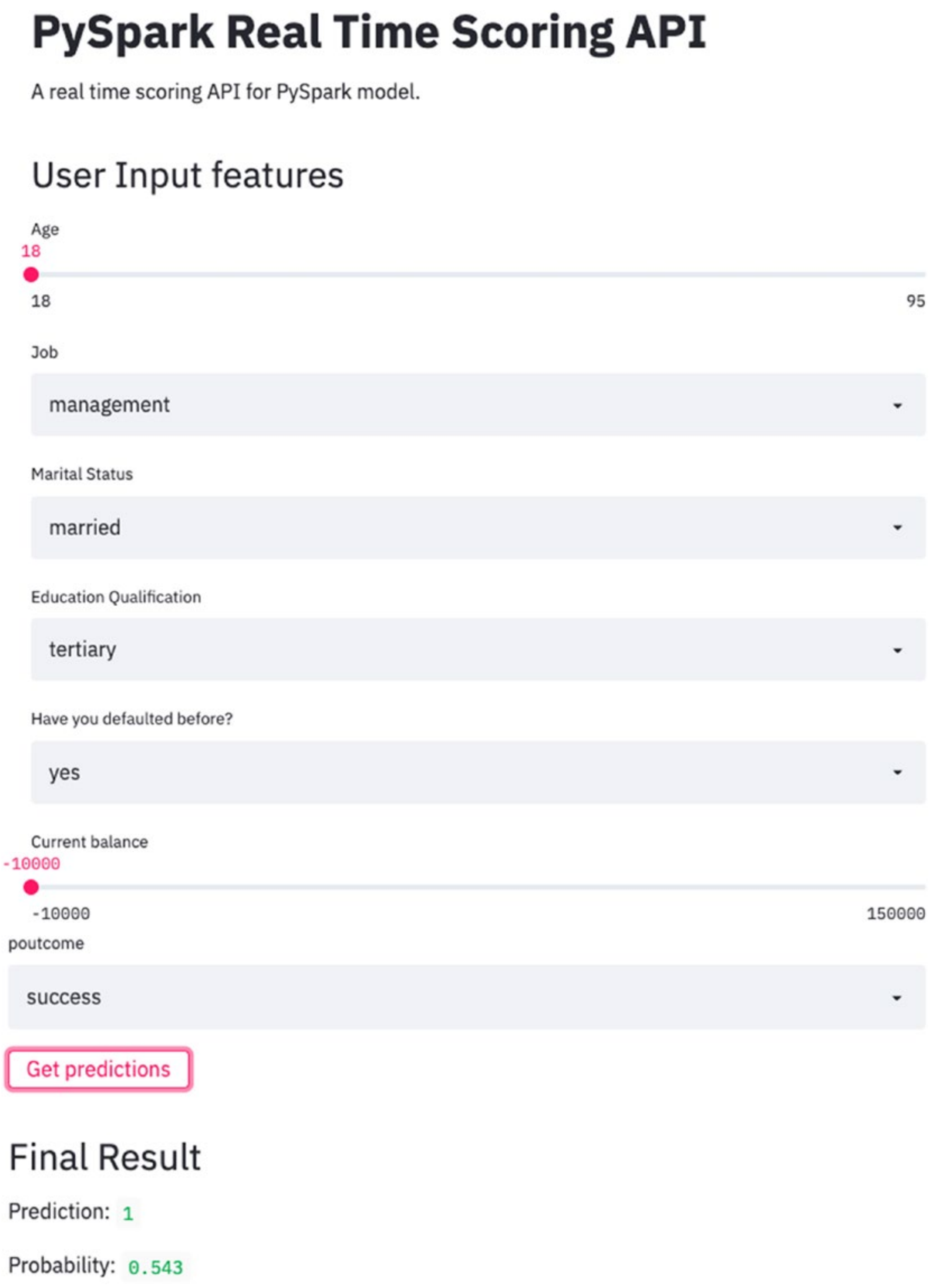
Figure 9-12. *Docker service creation*

Real-time Scoring API

To access the UI, you need to insert the following link in the browser:

```
http://localhost:8501/
```

The UI should look like Figure 9-13, if you used the same code as instructed in this book. We have shown only the partial UI in the figure. You can go ahead and input the values for the features and click *Get Predictions* to get results.


PySpark Real Time Scoring API

A real time scoring API for PySpark model.

User Input features

Age
18
18 95

Job
management

Marital Status
married

Education Qualification
tertiary

Have you defaulted before?
yes

Current balance
-10000
-10000 150000

poutcome
success

Final Result

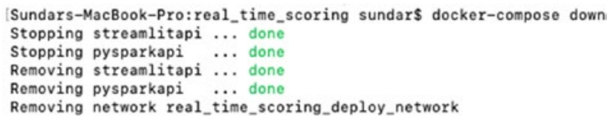
Prediction: 1

Probability: 0.543

Figure 9-13. *Streamlit API*

When you want to kill this service, you can use the following code (Figure 9-14):

```
docker-compose down
```



```
Sundars-MacBook-Pro:real_time_scoring sundar$ docker-compose down
Stopping streamlitapi ... done
Stopping pysparkapi   ... done
Removing streamlitapi ... done
Removing pysparkapi   ... done
Removing network real_time_scoring_deploy_network
```

Figure 9-14. *Killing Docker Service*

That's it. We have deployed our model in real-time with a nice front end. The best part of this deployment is that the UI and PySpark sit in separate Docker containers. They can be managed and customized separately and can be combined with a *docker-compose* file when needed. This makes the software development life-cycle process much easier.

Summary

- We went through the model deployment process in detail.
- We deployed a model using both batch and real-time processing.
- Finally, we created a custom UI front end using the `streamlit` API and combined with our PySpark API service using *docker-compose*.

Great job! In the next chapter, you will learn about experimentations and PySpark code optimizations. Keep learning and stay tuned.