4 Deep Learning for Predictive Maintenance

Predictive maintenance is one of the most sought after machine learning solutions for IoT. It is also one of the most elusive machine learning solutions for IoT. Other areas of machine learning can easily be solved, implementing Computer Vision, for example, can be done in hours using tools such as OpenCV or Keras. To be successful with predictive maintenance you first need the right sensors. The *Data collection design* recipe in `Chapter 2`, *Handling Data*, can be used to help determine proper sensor placement. The *Exploratory factor analysts* recipe in `Chapter 2`, *Handling Data* can help determine the cadence with which the data needs to be stored. One of the biggest hurdles to implementing predictive maintenance is that there needs to be a sufficient amount of device failures. For rugged industrial devices, this can take a long time. Linking repair records with device telemetry is also a critical step.

Even though the challenge is daunting the rewards are great. A properly implemented predictive maintenance solution can save lives by helping to ensure critical devices are ready when needed. They can also increase customer loyalty because they help companies have less downtime than similar products on the market. Finally, they can reduce costs and improve efficiency by giving service technicians the information they need before servicing the device. This can help them diagnose the device and ensure that they have the right parts with them when they are servicing the device.

In this chapter, we will continue to use the NASA Turbofan dataset for predictive maintenance and cover the following recipes:

- Enhancing data using feature engineering
- Using Keras for fall detection
- Implementing LSTM to predict device failure
- Deploying models to web services

Enhancing data using feature engineering

One of the best use of time in improving models is feature engineering. The ecosystem of IoT has many tools that can make it easier. Devices can be geographically connected or hierarchically connected with digital twins, graph frames, and GraphX. This can add features such as showing the degree of contentedness to other failing devices. Windowing can show how the current reading differs over a period of time. Streaming tools such as Kafka can combine different data streams allowing you to combine data from other sources. Machines that are outdoor may be negatively affected by high temperatures or moisture as opposed to machines that are in a climate-controlled building.

In this recipe, we are going to look at enhancing our data by looking at time-series data such as deltas, seasonality, and windowing. One of the most valuable uses of time for a data scientist is feature engineering. Being able to slice the data into meaningful features can greatly increase the accuracy of our models.

Getting ready

In the *Predictive maintenance with XGBoost* recipe in the previous chapter, we used XGBoost to predict whether or not a machine needed maintenance. We have imported the NASA *Turbofan engine degradation simulation* dataset which can be found at `https://data.nasa.gov/dataset/Turbofan-engine-degradation-simulation-data-set/vrks-gjie`. In the rest of this chapter, we will continue to use that dataset. To get ready you will need the dataset.

Then if you have not already imported `numpy`, `pandas`, `matplotlib`, and `seaborn` into Databricks do so now.

How to do it...

The following steps need to be observed to follow this recipe:

1. Firstly, import the required libraries. We will be using `pyspark.sql`, `numpy`, and `pandas` for data manipulation and `matplotlib` and `seaborn` for visualization:

```
from pyspark.sql import functions as F
from pyspark.sql.window import Window

import pandas as pd
import numpy as np
np.random.seed(1385)
```

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Next, we're going to import the data and apply a schema to it so that the data types can be correctly used. To do this we import the data file through the wizard and then apply our schema to it:

```
file_location = "/FileStore/tables/train_FD001.txt"
file_type = "csv"

from pyspark.sql.types import *
schema = StructType([
  StructField("engine_id", IntegerType()),
  StructField("cycle", IntegerType()),
  StructField("setting1", DoubleType()),
  StructField("setting2", DoubleType()),
  StructField("setting3", DoubleType()),
  StructField("s1", DoubleType()),
  StructField("s2", DoubleType()),
  StructField("s3", DoubleType()),
  StructField("s4", DoubleType()),
  StructField("s5", DoubleType()),
  StructField("s6", DoubleType()),
  StructField("s7", DoubleType()),
  StructField("s8", DoubleType()),
  StructField("s9", DoubleType()),
  StructField("s10", DoubleType()),
  StructField("s11", DoubleType()),
  StructField("s12", DoubleType()),
  StructField("s13", DoubleType()),
  StructField("s14", DoubleType()),
  StructField("s15", DoubleType()),
  StructField("s16", DoubleType()),
  StructField("s17", IntegerType()),
  StructField("s18", IntegerType()),
  StructField("s19", DoubleType()),
  StructField("s20", DoubleType()),
  StructField("s21", DoubleType())
  ])
```

3. Finally, we put it into a Spark DataFrame:

```
df = spark.read.option("delimiter"," ").csv(file_location,
                                            schema=schema,
                                            header=False)
```

4. We then create a temporary view so that we can run a Spark SQL job on it:

```
df.createOrReplaceTempView("raw_engine")
```

5. Next, we calculate **remaining useful life** (**RUL**). Using the SQL magics, we create a table named `engine` from the `raw_engine` temp view we just created. We then use SQL to calculate the RUL:

```
%sql

drop table if exists engine;

create table engine as
(select e.*
,mc - e.cycle as rul
, CASE WHEN mc - e.cycle < 14 THEN 1 ELSE 0 END as
needs_maintenance
from raw_engine e
join (select max(cycle) mc, engine_id from raw_engine group by
engine_id) m
on e.engine_id = m.engine_id)
```

6. We then import the data into a Spark DataFrame:

```
df = spark.sql("select * from engine")
```

7. Now we calculate the r**ate of change** (**ROC**). In the ROC calculation, we are looking at the ROC based on the current record compared to the previous record. The ROC calculation gets the percent of change between the current cycle and the previous one:

```
my_window = Window.partitionBy('engine_id').orderBy("cycle")
df = df.withColumn("roc_s9",
                   ((F.lag(df.s9).over(my_window)/df.s9) -1)*100)
df = df.withColumn("roc_s20",
                   ((F.lag(df.s20).over(my_window)/df.s20) -1)*100)
df = df.withColumn("roc_s2",
                   ((F.lag(df.s2).over(my_window)/df.s2) -1)*100)
df = df.withColumn("roc_s14",
                   ((F.lag(df.s14).over(my_window)/df.s14) -1)*100)
```

8. Next, we review static columns. In order to do that, we're going to convert the Spark DataFrame to Pandas so that we can view summary statistics on the data such as mean quartiles and standard deviation:

```
pdf = df.toPandas()
pdf.describe().transpose()
```

This will get the following output:

	count	mean	std	min	25%	50%	75%	max
engine_id	20631.0	51.506568	2.922763e+01	1.000000	26.000000	52.000000	77.000000	100.000000
cycle	20631.0	108.807862	6.888099e+01	1.000000	52.000000	104.000000	156.000000	362.000000
setting1	20631.0	-0.000009	2.187313e-03	-0.008700	-0.001500	0.000000	0.001500	0.008700
setting2	20631.0	0.000002	2.930621e-04	-0.000600	-0.000200	-0.000000	0.000300	0.000600
setting3	20631.0	100.000000	0.000000e+00	100.000000	100.000000	100.000000	100.000000	100.000000
s1	20631.0	518.670000	0.000000e+00	518.670000	518.670000	518.670000	518.670000	518.670000
s2	20631.0	642.680934	5.000533e-01	641.210000	642.325000	642.640000	643.000000	644.530000
s3	20631.0	1590.523119	6.131150e+00	1571.040000	1586.260000	1590.100000	1594.380000	1616.910000
s4	20631.0	1408.933782	9.000605e+00	1382.250000	1402.360000	1408.040000	1414.555000	1441.490000
s5	20631.0	14.620000	1.776400e-15	14.620000	14.620000	14.620000	14.620000	14.620000
s6	20631.0	21.609803	1.388985e-03	21.600000	21.610000	21.610000	21.610000	21.610000
s7	20631.0	553.367711	8.850923e-01	549.850000	552.810000	553.440000	554.010000	556.060000

9. Now we drop the columns that are not valuable to us in this exercise. For example, we are going to drop `settings3` and `s1` columns because the values never change:

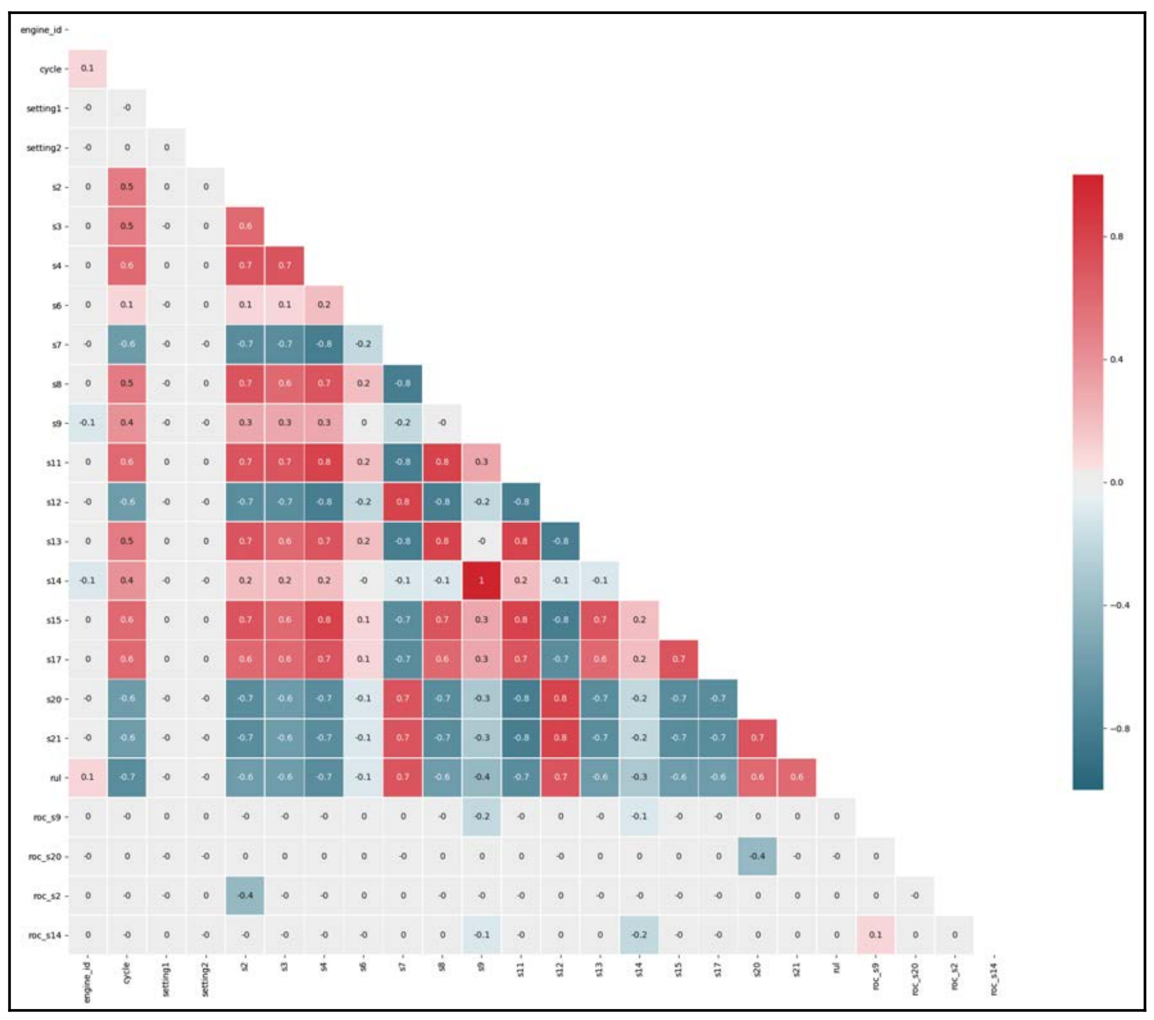
```
columns_to_drop = ['s1', 's5', 's10', 's16', 's18', 's19',
                   'op_setting3', 'setting3']
df = df.drop(*columns_to_drop)
```

10. Next, we are going to review the correlation between values. We are looking for columns that are exactly the same. First, we perform a correlation function on the DataFrame. Then we use `np.zeros_like` to mask the upper triangle. We are then going to set the figure size. Next, we are going to use `diverging_palette` to define a custom color map, then we are going to use the `heatmap` function do draw the heat map:

```
corr = pdf.corr().round(1)
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

f, ax = plt.subplots(figsize=(20, 20))

cmap = sns.diverging_palette(220, 10, as_cmap=True)

sns.heatmap(corr, mask=mask, cmap=cmap, vmin=-1, vmax=1, center=0,
```

```
                        square=True, linewidths=.5, cbar_kws={"shrink": .5},
                        annot=True)

display(plt.tight_layout())
```

The following heat map shows values with a high degree of correlation. The values that are `1` show that they are perfectly correlated and therefore can be dropped from the analysis:



11. Remove similar columns. We found that `S14` is exactly the same as `S9` so we are removing that column:

```
columns_to_drop = ['s14']
df = df.drop(*columns_to_drop)
```

12. Now we take the DataFrame and express it visually. A histogram or distribution table is used to show potential issues with our data such as outliers, skew data, random data and data that would not affect the model:

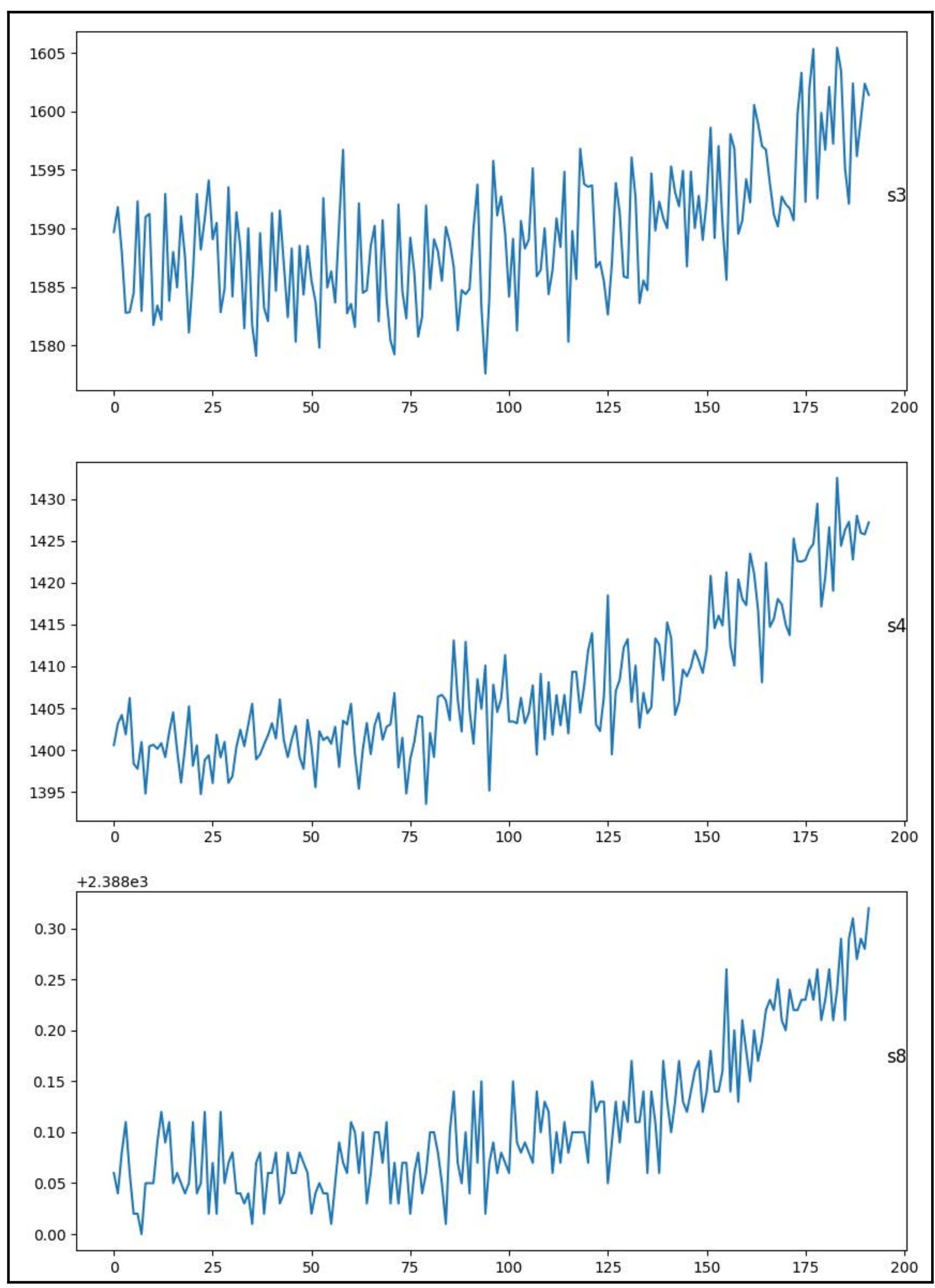
```
pdf = df.toPandas()

plt.figure(figsize = (16, 8))
plt.title('Example temperature sensor', fontsize=16)
plt.xlabel('# Cycles', fontsize=16)
plt.ylabel('Degrees', fontsize=16)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
pdf.hist(bins=50, figsize=(18,16))
display(plt.show())
```

The following histogram screenshots are the results:

13. We then review the noise of our model to make sure that it is not unduly affected by fluctuation:

```
values = pdf[pdf.engine_id==1].values
groups = [5, 6, 7, 8, 9, 10, 11,12,13]
i = 1
plt.figure(figsize=(10,20))
for group in groups:
    plt.subplot(len(groups), 1, i)
    plt.plot(values[:, group])
    plt.title(pdf.columns[group], y=0.5, loc='right')
    i += 1
display(plt.show())
```

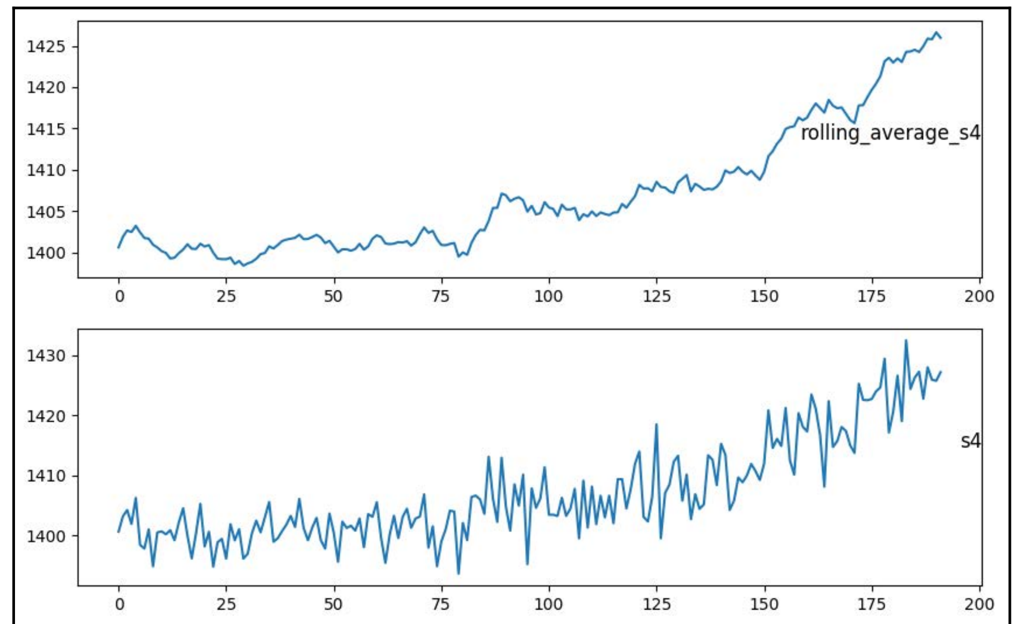
The following is the output:



14. Based on the previous step, it is clear that the data is noisy. This can lead to false readings. A rolling average can help smooth the data. Using a 7 cycle rolling average we denoise the data as shown:

```
w = (Window.partitionBy('engine_id').orderBy("cycle")\
     .rangeBetween(-7,0))
df = df.withColumn('rolling_average_s2', F.avg("s2").over(w))
df = df.withColumn('rolling_average_s3', F.avg("s3").over(w))
df = df.withColumn('rolling_average_s4', F.avg("s4").over(w))
df = df.withColumn('rolling_average_s7', F.avg("s7").over(w))
df = df.withColumn('rolling_average_s8', F.avg("s8").over(w))

pdf = df.toPandas()
values = pdf[pdf.engine_id==1].values
groups = [5, 25, 6, 26, 8, 27]
i = 1
plt.figure(figsize=(10,20))
for group in groups:
    plt.subplot(len(groups), 1, i)
    plt.plot(values[:, group])
    plt.title(pdf.columns[group], y=0.5, loc='right')
    i += 1
display(plt.show())
```

The following screenshot is a chart of `rolling_average_s4` versus `s4`:



15. Since we want this data to be accessible to other notebooks, we're going to save it as an ML ready table:

```
df.write.mode("overwrite").saveAsTable("engine_ml_ready")
```

How it works...

In this recipe, we have performed feature engineering so that we could make our data more usable by our ML algorithms. We removed the columns with no variation, high correlation, and we denoised the dataset. In *step 8* we removed the columns with no variation. The method describes the data in several ways. Reviewing the chart showed that many variables do not change at all. Next, we used a heat map to find sensors that had the same data. Finally, we used a rolling average to smooth the data from our original dataset into a new one.

There's more...

So far we have just looked at training data. But we will also need to look at testing the data. There is a test dataset and a RUL dataset. These datasets will help us test our models. To import them you would run 2 additional import steps:

1. **Importing test data**: Relying on the schema from the training set the test set is imported and put in a table called `engine_test`:

```
# File location and type
file_location = "/FileStore/tables/test_FD001.txt"
df = spark.read.option("delimiter"," ").csv(file_location,
                                            schema=schema,
                                            header=False)
df.write.mode("overwrite").saveAsTable("engine_test")
```

2. **Importing the RUL Dataset**: The next step is to import the remaining useful life dataset and save that to a table as well:

```
file_location = "/FileStore/tables/RUL_FD001.txt"
RULschema = StructType([StructField("RUL", IntegerType())])
df = spark.read.option("delimiter"," ").csv(file_location,
                                            schema=RULschema,
                                            header=False)
df.write.mode("overwrite").saveAsTable("engine_RUL")
```

Using keras for fall detection

One strategy for predictive maintenance is to look at patterns of device failures for a given record. In this recipe, we will classify the data that exhibits a pattern that happens before the device fails.

We will be using `keras`, which is a fairly powerful machine learning library. Keras strips away some of the complexity of TensorFlow and PyTorch. Keras is a great framework for beginners in machine learning as it is easy to get started on and the concepts learned in Keras transfer to more expressive machine learning libraries such as TensorFlow and PyTorch.

Getting ready

This recipe expands on the predictive maintenance dataset we feature engineered in the previous recipe. If you have not already done so you will need to import the `keras`, `tensorflow`, `sklearn`, `pandas`, and `numpy` libraries into your Databricks cluster.

How to do it...

Please observe the following steps:

1. Firstly, import the required libraries. We import `pandas`, `pyspark.sql`, and `numpy` for data manipulation, `keras` for machine learning, and `sklearn` for evaluating the model. After evaluating the model we use `io`, `pickle`, and `mlflow` to save the model and results so that it can be evaluated against other models:

```
from pyspark.sql.functions import *
from pyspark.sql.window import Window

import pandas as pd
import numpy as np
import io
import keras
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score
from sklearn.preprocessing import MinMaxScaler

from keras.models import Sequential
from keras.layers import Dense, Activation, LeakyReLU, Dropout
```

```
import pickle
import mlflow
```

2. Next, we import training and testing data. Out training data will be used to train our models and our testing data will be used to evaluate the models:

```
X_train = spark.sql("select rolling_average_s2, rolling_average_s3,
                    rolling_average_s4, rolling_average_s7,
                    rolling_average_s8 from \
                    engine_ml_ready").toPandas()

y_train = spark.sql("select needs_maintenance from \
                    engine_ml_ready").toPandas()

X_test = spark.sql("select rolling_average_s2, rolling_average_s3,
                    rolling_average_s4, rolling_average_s7,
                    rolling_average_s8 from \
                    engine_test_ml_ready").toPandas()

y_test = spark.sql("select needs_maintenance from \
                    engine_test_ml_ready").toPandas()
```

3. Now we scale the data. Each sensor of the dataset has a different scale. For example, the maximum value of `S1` is `518` while the maximum value of `S16` is `0.03`. For that reason, we convert all of the values to a range between `0` and `1`. Allowing each metric affect the model in a similar way. We will make use of the `MinMaxScaler` function from the `sklearn` library to adjust the scale:

```
scaler = MinMaxScaler(feature_range=(0, 1))
X_train.iloc[:,1:6] = scaler.fit_transform(X_train.iloc[:,1:6])
X_test.iloc[:,1:6] = scaler.fit_transform(X_test.iloc[:,1:6])
dim = X_train.shape[1]
```

4. The first layer, the input layer, has 32 nodes. The activation function, `LeakyReLU`, defines the output node when given the input. To prevent overfitting, 25% of the layers both hidden and visible are dropped when training:

```
model = Sequential()
model.add(Dense(32, input_dim = dim))
model.add(LeakyReLU())
model.add(Dropout(0.25))
```

5. Similar to the input layer, the hidden layer, uses 32 nodes as the input layer and `LeakyReLU` as its output layer. It also uses a 25% drop out to prevent overfitting:

```
model.add(Dense(32))
model.add(LeakyReLU())
model.add(Dropout(0.25))
```

6. Finally, we add an output layer. We give it one layer so that we can have an output between `0` and `1`. `sigmoid`, our activation function, helps predict the probability of the output. Our optimizer, `rmsprop`, along with the loss function helps optimize the data pattern and reduce the error rate:

```
model.add(Dense(1))
model.add(Activation('sigmoid'))
model.compile(optimizer ='rmsprop', loss ='binary_crossentropy',
              metrics = ['accuracy'])
```

7. Now we train the Model. We use the `model.fit` function to specify our training and test data. The batch size is used to set the number of training records used in 1 iteration of the algorithm. The epoch of `5` means that it will pass through the data set 5 times:

```
model.fit(X_train, y_train, batch_size = 32, epochs = 5,
          verbose = 1, validation_data = (X_test, y_test))
```

8. The next step is to evaluate the results. We use the trained model and our `X_test` dataset to get the predictions (`y_pred`). We then compare the predictions with the real results and review how accurate it is:

```
y_pred = model.predict(X_test)
pre_score = precision_score(y_test,y_pred, average='micro')
print("Neural Network:",pre_score)
```

9. Next, we save the results to `mlflow`. The results will be compared against the other ML algorithms for predictive maintenance we are using in this book:

```
with mlflow.start_run():
mlflow.set_experiment("/Shared/experiments/Predictive_Maintenance")
    mlflow.log_param("model", 'Neural Network')
    mlflow.log_param("Inputactivation", 'Leaky ReLU')
    mlflow.log_param("Hiddenactivation", 'Leaky ReLU')
    mlflow.log_param("optimizer", 'rmsprop')
    mlflow.log_param("loss", 'binary_crossentropy')
```

```
mlflow.log_metric("precision_score", pre_score)
filename = 'NeuralNet.pickel'
pickle.dump(model, open(filename, 'wb'))
mlflow.log_artifact(filename)
```

How it works...

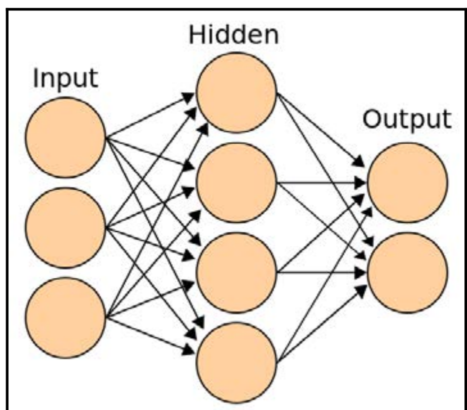
There are typically three tasks that neural networks does:

- Import data
- Recognize the patterns of the data by training
- Predicting the outcomes of new data

Neural networks take in data, trains themselves to recognize the patterns of the data, and then are used to predict the outcomes of new data. This recipe uses the cleaned and feature engineered dataset saved in the previous recipe. The `X_train` dataset is pulled in from the `spark` data table into a Panda DataFrame. The training DataFrames, `X_train`, and `y_train` are used for training. `X_test` gives us a list of devices that have failed and `y_test` gives us the real-time failure of those machines. Those datasets are used to train models and test the results.

First, we have the input layer. The data is fed to each of our 32 input neurons. The neurons are connected through channels. The channel is assigned a numerical value known as **weight**. The inputs are multiplied by the corresponding weight and their sum is sent as input to the neurons in the hidden layer. Each of these neurons is associated with a numerical value called the **bias**, which is added to the input sum. This value is then passed to a threshold function called the **activation function**. The activation function determines if a neuron will get activated or not. We used Leaky ReLU as our activation function for our first 2 layers. **ReLU** or **Rectified Linear Unit** is a popular activation function because it solves the vanishing gradient problem. In this recipe, we used the Leaky ReLU. Leaky ReLU solves a problem that ReLU has where big gradients can cause the neuron to never fire. The activated neuron passes its data to the next layer over the channels. This method allows the data to be propagated through the network. This is called **forward propagation**. In the output layer, the neuron with the highest layer fires and determines the output.

When we first start running data through our network, the data usually has a high degree of error. Our error and optimizer functions use backpropagation to update the weights. The cycle of forward propagation and backpropagation is repeated to achieve a lower error rate. The following diagram shows how the input, hidden, and output layers are linked together:



There's more...

In this recipe, we used `LeakyReLU` as our activation function, `rmsprop` as our optimizer, and `binary_crossentropy` as our loss function. We then saved the results to `mlflow`. We can tune parameters in this experiment by trying different combinations such as the number of neurons or the number of layers. We could also change the activation function to use ReLU or TanH. We could also use `Adam` as our optimizer. Saving those results to `mlflow` allows us to improve our model.

Implementing LSTM to predict device failure

Recurrent neural networks predict sequences of data. In the previous recipe, we looked at 1 point in time and determined to determine if maintenance was needed. As we saw in the first recipe when we did the data analysis the turbofan run to failure dataset is highly variable. The data reading at any point in time might indicate a need for maintenance while the next indicates that there is no need for maintenance. When determining whether or not to send a technician out having an oscillating signal can be problematic. **Long Short Term Memory** (**LSTM**) is often used with time-series data such as the turbofan run to failure dataset.

With the LSTM, we look at a series of data, similar to windowing. LSTM uses an ordered sequence to help determine, in our case, if a turbofan engine is about to fail based on the previous sequence of data.

Getting ready

For this recipe we will use the NASA *Turbofan run to failure* dataset. For this recipe we will be using a Databricks notebook. This recipe requires a few libraries to be installed. For data processing we need to install `numpy` and `pandas`, `keras` for creating a LSTM model, and `sklearn` and `mlflow` for evaluating and saving the results of our model.

Even though in previous recipes we added windowing and preprocessed the data, in this recipe we will use the raw data. LSTMs window the data and also have a good deal of extraction and transformation that is unique to this type of ML Algorithm.

How to do it...

We will execute the following steps for this recipe:

1. First, we will import all of the libraries which we will need later. We will import `pandas` and `numpy` for data processing, `keras` for the ML models, `sklearn` for evaluations, and `pickel` and `mlflow` for storing the results:

```
import pandas as pd
import numpy as np

import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, LSTM, Activation

from sklearn import preprocessing
from sklearn.metrics import confusion_matrix, recall_score,
precision_score
import pickle
import mlflow
```

2. Next we will set the variables. We will set 2 cycles periods. In addition we use a sequence length variable. The sequence length allows the LSTM to look back over 5 cycles. This is similar to windowing that was discussed in `Chapter 1`, *Setting Up the IoT and AI Environment*. We are also going to get a list of data columns:

```
week1 = 7
week2 = 14
sequence_length = 100
sensor_cols = ['s' + str(i) for i in range(1,22)]
sequence_cols = ['setting1', 'setting2', 'setting3', 'cycle_norm']
sequence_cols.extend(sensor_cols)
```

3. Next we import data from the `spark` data tables we created in the *Simple predictive maintenance with XGBoost* recipe in `Chapter 3`, *Machine Learning for IoT*. We also drop the `label` column because we are going to recalculate the labels. We are going to import three DataFrames. The `train` DataFrame is used to train the model. The `test` DataFrame is used to test the accuracy of the model and the `truth` DataFrame is the actual failures for the `test` DataFrame:

```
train = spark.sql("select * from engine").toPandas()
train.drop(columns="label" , inplace=True)
test = spark.sql("select * from engine_test2").toPandas()
truth = spark.sql("select * from engine_rul").toPandas()
```

4. Then, we generate labels that show if a device needs maintenance. `label1` shows when a device will fail in 14 cycles and `label2` shows when a device will fail in 7 cycles. First we create a DataFrame that shows the RUL based on the maximum cycle number for each engine. Next we use that the RUL DataFrame create a RUL column in our train DataFrame. We do this by subtracting the maximum life from the current cycle. We then drop our `max` column. Next we create a new column `label1`. `label1` has a `1` value if the RUL is less than the 14 cycles. Then copy that over to `label2` and add a `2` value if the RUL is less than 1 week:

```
rul = pd.DataFrame(train.groupby('engine_id')['cycle']\
                   .max()).reset_index()
rul.columns = ['engine_id', 'max']
train = train.merge(rul, on=['engine_id'], how='left')
train['RUL'] = train['max'] - train['cycle']
train.drop('max', axis=1, inplace=True)
train['label1'] = np.where(train['RUL'] <= week2, 1, 0 )
train['label2'] = train['label1']
train.loc[train['RUL'] <= week1, 'label2'] = 2
```

5. In addition to generating labels for training data we also need to do so for test data. The training and test data are slightly different. The training data had an end date that signified when the machine broke. The training set does not. Instead we have a `truth` DataFrame that shows when the machine actually failed. To add the label columns we need to combine the `test` and `truth` dataset before we can calculate the labels:

```
rul = pd.DataFrame(test.groupby('engine_id')['cycle'].max())\
                   .reset_index()
rul.columns = ['engine_id', 'max']
truth.columns = ['more']
truth['engine_id'] = truth.index + 1
truth['max'] = rul['max'] + truth['more']
truth.drop('more', axis=1, inplace=True)

test = test.merge(truth, on=['engine_id'], how='left')
test['RUL'] = test['max'] - test['cycle']
test.drop('max', axis=1, inplace=True)

test['label1'] = np.where(test['RUL'] <= week2, 1, 0 )
test['label2'] = test['label1']
test.loc[test['RUL'] <= week1, 'label2'] = 2
```

6. Next, because the columns have different mins and maxes, we will normalize the data so that one variable does not overshadow the rest. To do this we are going to use the `sklearn` library's `MinMaxScaler` function. This function transforms the values between `0` and `1`. It is a great scalier to use when, as in our case, there is not a lot of outlier values. We are going to do the same normalization step for both the training and test set:

```
train['cycle_norm'] = train['cycle']
cols_normalize =
train.columns.difference(['engine_id','cycle','RUL',
                                            'label1','label2'])
min_max_scaler = preprocessing.MinMaxScaler()
norm_train = \
pd.DataFrame(min_max_scaler.fit_transform(train[cols_normalize]),
                                          columns=cols_normalize,
                                          index=train.index)
join = \
train[train.columns.difference(cols_normalize)].join(norm_train)
train = join.reindex(columns = train.columns)

test['cycle_norm'] = test['cycle']
norm_test = \
pd.DataFrame(min_max_scaler.transform(test[cols_normalize]),
```

```
                                              columns=cols_normalize,
                                              index=test.index)
test_join = \
test[test.columns.difference(cols_normalize)].join(norm_test)
test = test_join.reindex(columns = test.columns)
test = test.reset_index(drop=True)
```

7. The LSTM algorithm in Keras requires the data to be in a sequence. In our variables section, we chose to have `sequence_length` equal to `100`. This is one of the hyperparameters that can be tuned during experimentation. As this is a look at data over a sequential period of time the sequence length is the length of the sequence of data we are training the model on. There is no real rule of thumb on what is the optimal length for a sequence. But from experimentation, it became clear that small sequences were less accurate. To aid in generating our sequence we use the function to return the sequential data in a way that the LSTM algorithm expects:

```
def gen_sequence(id_df, seq_length, seq_cols):
    data_array = id_df[seq_cols].values
    num_elements = data_array.shape[0]
    for start, stop in zip(range(0, num_elements-seq_length),
                           range(seq_length, num_elements)):
        yield data_array[start:stop, :]

seq_gen = (list(gen_sequence(train[train['engine_id']==engine_id],
                             sequence_length, sequence_cols))
           for engine_id in train['engine_id'].unique())

seq_array = np.concatenate(list(seq_gen)).astype(np.float32)
```

8. The next step is to build a neural network. We build the first layer of our LSTM. We start off with a sequential model. Then give it the input shape and length of the sequence. The units tell us the dimensionality of the output shape which it will pass to the next layer. Next, it returns either `true` or `false`. We then add `Dropout` to add the randomness to our training that prevents overfitting:

```
nb_features = seq_array.shape[2]
nb_out = label_array.shape[1]

model = Sequential()

model.add(LSTM(input_shape=(sequence_length, nb_features),
               units=100, return_sequences=True))
model.add(Dropout(0.25))
```

9. We then build the network's hidden layer. Similar to the first layer the hidden layer is an LSTM layer. If, however, instead of passing the entire sequence state to the output just passes the last nodes' values:

```
model.add(LSTM(units=50, return_sequences=False))
model.add(Dropout(0.25))
```

10. Then, we build the network's output layer. The output layer specifies the output dimensions and the `activation` function. With this we have built the shape of our neural network:

```
model.add(Dense(units=nb_out, activation='sigmoid'))
```

11. Next, we run the `compile` method which configures the model for training. In it we put the metric we are evaluating against. In this case, we are measuring against `accuracy`. We then define our measure of error or loss. In this example, we are using `binary_crossentropy` as our measure. Finally, we specify the optimizer that will reduce error every iteration:

```
model.compile(loss='binary_crossentropy', optimizer='adam',
              metrics=['accuracy'])
print(model.summary())
```

12. We then use our `fit` function to train the model. Our `epochs` parameters means that the data will be run through 10 times. Because of the random dropout, the extra runs will increase accuracy. We are using `batch_size` of `200`. This means that model will train through 200 samples before it updates the gradients. Next, we use `validation_split` to put 95% of the data to training the model and 5% to validating the model. Finally, we use an `EarlyStopping` callback to stop the model from training when it stops improving accuracy:

```
model.fit(seq_array, label_array, epochs=10, batch_size=200,
          validation_split=0.05, verbose=1,
          callbacks = \
          [keras.callbacks.EarlyStopping(monitor='val_loss',
                                         min_delta=0, patience=0,
                                         verbose=0, mode='auto')])
```

13. Next, we evaluate our model based on the 95%/5% split we performed on the training data. The results show our model is evaluating the 5% data that we held back at an 87% accuracy:

```
scores = model.evaluate(seq_array, label_array, verbose=1,
                        batch_size=200)
print('Accuracy: {}'.format(scores[1]))
```

14. Next, we look at the confusion matrix which shows us a matrix of correct or wrong assessments of whether the engine needed maintenance or not:

```
y_pred = model.predict_classes(seq_array,verbose=1, batch_size=200)
y_true = label_array
print('Confusion matrix\n- x-axis is true labels.\n- y-axis is
predicted labels')
cm = confusion_matrix(y_true, y_pred)
cm
```

Our confusion matrix looks like the following grid:

	Actually Did not need maintenance	**Predicted Needed Maintenance**
Actually Did not need maintenance	13911	220
Actually Needed Maintenance	201	1299

15. We then compute the precision and recall. Because the dataset is unbalanced, meaning there are far more values that do not need maintenance than they do, precision and recall are the most appropriate measures for evaluating this algorithm:

```
precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
print( 'precision = ', precision, '\n', 'recall = ', recall)
```

16. Next, we need to transform the data so that the testing data is the same type of sequential data that the training data is. To do this we perform a data transformation step similar to the one we did for the training data:

```
seq_array_test_last = [test[test['engine_id']==engine_id]\
[sequence_cols].values[-sequence_length:] for engine_id in \
test['engine_id'].unique() if \
len(test[test['engine_id']==engine_id]) >= sequence_length]

seq_array_test_last = \
np.asarray(seq_array_test_last).astype(np.float32)

y_mask = [len(test[test['engine_id']==engine_id]) >= \
          sequence_length for engine_id in \
          test['engine_id'].unique()]

label_array_test_last = \
test.groupby('engine_id')['label1'].nth(-1)[y_mask].values
label_array_test_last = label_array_test_last.reshape(
    label_array_test_last.shape[0],1).astype(np.float32)
```

17. Next, we evaluate the model generated with the training dataset against the test dataset to see how accurately the model predicts when an engine needs maintenance:

```
scores_test = model.evaluate(seq_array_test_last,
                             label_array_test_last, verbose=2)
print('Accuracy: {}'.format(scores_test[1]))
y_pred_test = model.predict_classes(seq_array_test_last)
y_true_test = label_array_test_last
print('Confusion matrix\n- x-axis is true labels.\n- y-axis is
predicted labels')
cm = confusion_matrix(y_true_test, y_pred_test)
print(cm)

pre_score = precision_score(y_true_test, y_pred_test)
recall_test = recall_score(y_true_test, y_pred_test)
f1_test = 2 * (pre_score * recall_test) / (pre_score + recall_test)
print('Precision: ', pre_score, '\n', 'Recall: ', recall_test,
      '\n', 'F1-score:', f1_test )
```

18. Now that we have our results we store those along with the model in our MLflow database:

```
with mlflow.start_run():
mlflow.set_experiment("/Shared/experiments/Predictive_Maintenance")
    mlflow.log_param("type", 'LSTM')
    mlflow.log_metric("precision_score", pre_score)
    filename = 'model.sav'
    pickle.dump(model, open(filename, 'wb'))
    mlflow.log_artifact(filename)
```

How it works...

A LSTM is a special type of **recurrent neural network** (**RNN**). A RNN is a neural network architecture that deal with sequenced data by keeping the sequence in memory. Conversely, a typical feed-forward neural does not keep the information about the sequences and do not allow for flexible inputs and outputs. A recursive neural network uses recursion to call from one output back to its input thereby generating a sequence. It passes a copy of the state of the network at any given time. In our case we are using two layers for our RNN. This additional layer helps with accuracy.

LSTMs solve a problem of vanilla RNNs by dropping out data to solve the vanishing gradient problem. The vanishing gradient problem is when the neural network stops training early but is inaccurate. By using dropout data we can help solve that problem. The LSTM does this by using gating functions.

Deploying models to web services

Deployment of the model can be different depending on the capabilities of the device. Some devices with extra compute can handle having the machine learning models run directly on the device. While others require assistance. In this chapter, we are going to deploy the model to a simple web service. With modern cloud web apps or Kubernetes these web services can scale to meet the needs of the fleet of devices. In the next chapter, we will show how to run the model on the device.

Getting ready

So far in this book, we have looked at three different machine learning algorithms to solve the predictive maintenance problem with the NASA Turbofan run to failure dataset. We recorded the results to MLflow. We can see that our XGBoost notebook outperformed the more complex neural networks. The following screenshot shows the MLflow result set showing the parameters and their associated scores.

Source	Versi...	Tags	Parameters	Metrics
Predic...			type: XGBoost	precision_score: 0.972082202404...
Deep ...			type: LSTM	precision_score: 0.888888888888...
Deep ...			type: LSTM	precision_score: 0.888888888888...
Deep ...			type: LSTM	precision_score: 0.875

From here we can download our model and put it in our web service. To do this we are going to use a Python Flask web service and Docker to make the service portable. Before we start, `pip install` the python `Flask` package. Also install Docker onto your local computer. Docker is a tool that allows you to build out complex deployments.

How to do it...

In this project, you will need to create three files for testing the predictor web service and one file to scale it to production. First create `app.py` for our web server, `requirements.txt` for the dependencies, and the XGBoost model you downloaded from `mlflow`. These files will allow you to test the web service. Next, to put it into production you will need to dockerize the application. Dockerizing the file allow you to deploy it to services such as cloud-based web application or Kubernetes services. These services scale easily making onboarding new IoT devices seamless. Then execute the following steps:

1. The `app.py` file is the Flask application. Import `Flask` for the web service, `os` and `pickle` for reading the model into memory, `pandas` for data manipulation, and `xgboost` to run our model:

```
from flask import Flask, request, jsonify
import os
import pickle
import pandas as pd
import xgboost as xgb
```

2. Next is to initialize our variables. By loading the Flask application and XGBoost model into memory outside a function we ensure that it only loads once rather than on every web service call. By doing this we greatly increase the speed and efficiency of the web service. We use `pickle` to re-hydrate our model. `pickle` can take almost any Python object and write it to disk. It can also, as in our case, read if from disk and put it back into memory:

```
application = Flask(__name__)
model_filename = os.path.join(os.getcwd(), 'bst.sav')
loaded_model = pickle.load(open(model_filename, "rb"))
```

3. We then create `@application.route` to give us an `http` endpoint. The `POST` methods section specifies that it will only accept post web request. We also specify that the URL will route to `/predict`. For example, when we run this locally we could use the `http://localhost:8000/precict` URL to post our JSON string. We then convert it into a `pandas` DataFrame and then an XGBoost data matrix becomes calling `predict`. We then determine if it is above `.5` or below and return the results:

```
@application.route('/predict', methods=['POST'])
def predict():
    x_test = pd.DataFrame(request.json)
    y_pred = loaded_model.predict(xgb.DMatrix(x_test))
    y_pred[y_pred > 0.5] = 1
```

```
        y_pred[y_pred <= 0.5] = 0
        return int(y_pred[0])
```

4. Finally, the last thing to do in any Flask app is to call the `application.run` method. This method allows us to specify a host. In this case, we are specifying a special host of `0.0.0.0` which tells flask to accept requests from other computers. Next, we specify a port. The port can be any number. It does however need to match the port in the Dockerfile:

```
if __name__ == '__main__':
    application.run(host='0.0.0.0', port=8000)
```

5. We then create a requirements file. The `requirements.txt` file will install all of the python dependencies for the project. The docker will use this to install the dependencies:

```
flask
pandas
xgboost
pickle-mixin
gunicorn
```

6. Then, we create the Dockerfile. The `docker` file allows the deployment of the predictor to a web endpoint. The first line of the docker file will pull in the official python 3.7.5 image from Docker Hub. Next, we copy the local folder to a new folder in the docker container in a folder named `app`. Next, we set the working directory to the `app` folder. Then we use `pip install` to install the requirements from the file we created in *step 5*. Then we expose port `8000`. Finally, we run the `gunicorn` command that starts the Gunicorn server:

```
FROM python:3.7.5
ADD . /app
WORKDIR /app

RUN pip install -r requirements.txt

EXPOSE 8000
CMD ["gunicorn", "-b", "0.0.0.0:8000", "app"]
```

How it works...

Flask is a lightweight web server. We pull in the model that we saved to disk using `pickle` to rehydrate the model. We then create an `http` endpoint to call into.

There's more...

Modern cloud-based web applications such as **Azure Web Apps** can automatically pull new Docker images into production. There is also a great deal of DevOps tools that can pull images and run them through various tests before deploying them with Docker container instances or docker orchestration tools such as Kubernetes. But for them to do this one must first put them into a container registry such as **Azure Container Registry** or **Docker Hub**. To do this we will need to do a few steps. First, we will build our container. Next, we can run our container to ensure that it works. Then we log into our container registry service and push the container to it. The detailed steps are as follows:

1. First, we build the container. To do it we navigate to the folder with the docker file and run docker build. We are going to tag it with the `-t` command to `ch4`. We then specify that the docker file is in the local folder with the period `.`:

```
docker build -t ch4 .
```

2. Now that we have a docker image built, we are going to run the container based on the image with the `docker run` command. We are going to use the `-it` interactive command so we can see any output from the server. We are also going to use the `-p` or `port` command to specify that we are mapping the docker containers internal port `8000` to the external port `8000`:

```
docker run -it -p 8000:8000 ch4
```

3. We then need to put the container into something that can be accessible by our compute resource. To do this, first register for a Docker Registry service such as Docker Hub or Azure Container Registry. Then create a new repository. The repository provider will give you a path for that repository.
4. Next is to log in to your container registry service, tag the container, and push the container. Remember to replace `[Your container path]` with the registry name or path provided by the registry service:

```
docker login

docker tag ch4 [Your container path]:v1
docker push [Your container path]:v1
```

You can then use docker enabled cloud technology to push that predictor service into production. Your device can then send its sensor reading to the web service and receive through a cloud to device message whether the device needs maintenance or not.