

CHAPTER 3

Machine Learning Deployment as a Web Service

In this chapter, we are going to go over how to use different web frameworks for deploying machine learning and deep learning models as web services hosted on the local system. This chapter covers three main topics.

The chapter first introduces the Flask framework and how to deploy an ML model using it. Then the chapter shows how to build a standard machine learning model and deploy it using another web framework, Streamlit. Finally, the chapter covers how to deploy a trained deep learning model using the Streamlit platform again. If you are already comfortable with Flask basics, feel free to skip to the deployment part of the first section to understand the process of deploying machine learning models using Flask. As mentioned in the previous chapter, we mainly use Jupyter notebooks to develop and test the models locally, but when we want to connect our ML model to an app or a web service, we essentially need to deploy it using a web server. This web server can be hosted locally or in the cloud. There are many different ways in which a machine learning model can be deployed, but in this chapter, we are going to explore two methods: Flask and Streamlit.

Introduction to Flask

In simple words, Flask is an open source lightweight web framework built in Python to deploy web applications. When we say *web framework*, we mean a group of resources needed to run a web application. This might include different modules, libraries, and tools that can be used by the web developer to successfully build and run the application. Unfortunately, this book does not do a deep dive into Flask, but for those of you who have never used it before, the following code snippet gives a quick introduction to Flask. To use Flask, we first need to install it on the local machine. We can simply use `pip install flask` to install Flask.

```
[In]: from flask import Flask

[In]: app = Flask(__name__)

[In]: @app.route("/")
[In]: def hello():
    return "Hello World!"

[In]: if __name__ == '__main__':
    app.run(debug=True)
```

route Function

The route function is a decorator that tells which URL is associated with a particular function. It has two parameters.

- rule
- options

The rule indicates the URL path and its binding with the given function. It renders the current output of the function when the URL is opened in the browser. The options let you pass different sets of parameters.

run Method

The run method executes the application to run on the particular web server. It has four parameters that can be passed during execution. All of these are optional parameters, and `app.run` can be executed without passing any of them as well.

- `host`
- `port`
- `debug`
- `options`

By default, the application runs on localhost (127.0.0.1). The debug option is set to false by default and can be set to true to see the debug information.

Deploying a Machine Learning Model as a REST Service

Now that we know how the Flask framework works, we are going to build a simple linear regression model and deploy it using the Flask server. We start by importing the required libraries.

```
[In]: import pandas as pd
[In]: import numpy as np
[In]: from sklearn.linear_model import LinearRegression
[In]: import joblib
```

We load the dataset into pandas, and as we can see, our dataset contains five input columns and a target column.

```
[In]: df=pd.read_csv('Linear_regression_dataset.
      csv',header='infer')
[In]: df.sample(5)
[Out]:
```

	var_1	var_2	var_3	var_4	var_5	output
876	664	724	79	0.331	0.264	0.361
1167	654	667	85	0.313	0.250	0.384
724	729	688	84	0.318	0.253	0.424
1222	842	697	102	0.337	0.268	0.400
482	772	771	76	0.320	0.251	0.401

Since the idea is not to build a super-powerful model but rather deploy an ML model, we need not split this data into train and test sets. We fit a linear regression model and get a decent r-square value.

```
[In]: X=df.loc[:,df.columns !='output']
[In]: y=df['output']
[In]: lr = LinearRegression().fit(X, y)
[In]: lr.score(X,y)
[Out]: 0.8692670151914198
```

The next step is to save the trained model that can be loaded back while serving it as a web service. We make use of the `joblib` library that serializes the model (saves coefficient values for input variables as a dictionary).

```
[In]: joblib.dump(lr,'inear_regression_model.pkl')
```

Now that we have saved the model, we can create the main `app.py` file, which will spin up the Flask server to run the ML model as a web app.

```
[In]: import pandas as pd
[In]: import numpy as np
[In]: import sklearn
[In]: import joblib
[In]: from flask import Flask,render_template,request
[In]: app=Flask(__name__)

[In]: @app.route('/')
```

```

[In]: def home():
        return render_template('home.html')

[In]: @app.route('/predict',methods=['GET','POST'])

[In]: def predict():
        if request.method == 'POST':
            print(request.form.get('var_1'))
            print(request.form.get('var_2'))
            print(request.form.get('var_3'))
            print(request.form.get('var_4'))
            print(request.form.get('var_5'))
            try:
                var_1=float(request.form['var_1'])
                var_2=float(request.form['var_2'])
                var_3=float(request.form['var_3'])
                var_4=float(request.form['var_4'])
                var_5=float(request.form['var_5'])
                pred_args=[var_1,var_2,var_3,var_4,var_5]
                pred_arr=np.array(pred_args)
                preds=pred_arr.reshape(1,-1)
                model=open("linear_regression_model.
                pkl","rb")
                lr_model=joblib.load(model)
                model_prediction=lr_model.predict(preds)
                model_prediction=round(float(model_
                prediction),2)
            except ValueError:
                return "Please Enter valid values"
        return render_template('predict.html',prediction=model_
        prediction)

[In]: if __name__=='__main__':
        app.run(host='0.0.0.0')

```

Let's go over the steps to understand the details of the `app.py` file. First, we import all the required libraries from Python. Next, we create our first function, which is the home page that renders the HTML template to allow the users to fill input values. The next function is publishing the predictions by the model on those input values provided by the user. We save the input values into five different variables coming from the user and create a list (`pred_args`). We then convert that into a numpy array. We reshape it into the desired form to be able to make a prediction on it. The next step is to load the trained model (`linear_regression_model.pkl`) and make the predictions. We save the final output into a variable (`model_prediction`). We then publish these results via another HTML template, `predict.html`. If we run the main file (`app.py`) now in the terminal, we would see the page come up asking the user to fill the values, as shown in Figure 3-1.

Prediction from Regression

Enter the values



The form consists of five vertically stacked input fields, each preceded by a label: `var_1`, `var_2`, `var_3`, `var_4`, and `var_5`. Below these fields is a blue rectangular button with the text "Submit" in white.

Figure 3-1. Input for ML prediction

Templates

There are two web pages that we have to design to post requests to the server and receive the response message that is the prediction by the ML model for that particular request. Since this book doesn't focus on HTML, you can simply use these files as is without making any changes to them, as shown in Figure 3-2. But for curious readers, we are creating a form to request five values in five different variables. We are using a standard CSS template with some basic fields. Users with prior knowledge of HTML can feel free to redesign the home page as per their requirements.

```
<!DOCTYPE html>
<html>
<head>
  <<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
  bootstrap/3.3.7/css/bootstrap.min.css" integrity="
  sha384-BVYiiSIFeK1dGm1RAkycuHAHRg320Mjcw7on3Ryd94v6+Pm5Sz/K68vbdEjh4u"
  crossorigin="anonymous">
  <title>Prediction Using Flask</title>
</head>
<body>
  <h1><div style="text-align:center"><font color='blue'>Prediction from
  Regression </font></div></h1>
  <br>
  <h2><div style="text-align:center">Enter the values </div></h2>
  </br>
  <div class="container">
    <div class="row">
      <div class="col-6">
        <form method="POST" action="/predict">
          <div class="form-group">
            <label form="var_1"><p class="font-weight-bold">
              var_1</p></label>
            <input type="text" name="var_1">
          </div>
          <div class="form-group">
            <label form="var_2"><p class="font-weight-bold">
              var_2</p></label>
            <input type="text" name="var_2">
          </div>
          <div class="form-group">
            <label form="var_3"><p class="
            font-weight-bold">var_3</p></label>
            <input type="text" name="var_3">
          </div>
          <div class="form-group">
            <label form="var_4"><p class="
            font-weight-bold">var_4</p></label>
            <input type="text" name="var_4">
          </div>
          <div class="form-group">
            <label form="var_5"><p class="
            font-weight-bold">var_5</p></label>
            <input type="text" name="var_5">
          </div>
          <input class="btn btn-primary" type
          ="submit" value="Submit">
        </form>
      </div>
    </div>
  </div>
</body>
</html>
```

Figure 3-2. Input request HTML form

The next template is to publish the model prediction to the user. This is less complicated compared to the first template as there is just one value that we have to post back to the user, as shown in Figure 3-3.

```
<!DOCTYPE html>
<html>
<head>
  <<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
bootstrap/3.3.7/css/bootstrap.min.css" integrity="
sha384-BVYiiSIFeK1dGmJRAkycuHAHRg320mUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u"
crossorigin="anonymous">

  <title>Prediction by ML model</title>
</head>

<body>
  <h1><div style="text-align:center"><font color='blue'>Prediction
Result </font></div></h1>
  <hr>
  <div class='card text-center' style="width:21.5em;margin:0 auto;">
    <div class="card-body">
      <p class="card text"><h1><font color='blue'>{{prediction}}</
font></h1></p>
    </div>
  </div>
</body>
</html>
```

Figure 3-3. Model prediction HTML form

Let's go ahead and input values for the model prediction, as shown in Figure 3-4. As we can observe in Figure 3-5, the model prediction result is a continuous variable since we have trained a regression model.

Prediction from Regression

Enter the values



The figure shows a web form with five input fields, each preceded by a label: var_1, var_2, var_3, var_4, and var_5. The values entered in the fields are 34728, 32123, 213123, 12312, and 23412 respectively. Below the fields is a blue 'Submit' button.

Label	Value
var_1	34728
var_2	32123
var_3	213123
var_4	12312
var_5	23412

Submit

Figure 3-4. User inputs page

Prediction Result

3608.45

Figure 3-5. Model prediction

As we saw, Flask makes it easy to deploy the machine learning app as a web service. One disadvantage of using Flask is that since it is a lightweight web framework, it has a limited capacity to handle complex applications. Another disadvantage is that most data scientists are not comfortable with using HTML and JavaScript to create the front end for the application. Hence, in the next section, we are going to look at a much simpler alternative of deploying a machine learning app using Streamlit. This makes it easier to develop a simple UI for the app compared to Flask.

Deploying a Machine Learning Model Using Streamlit

Streamlit is an alternative to Flask for deploying the machine learning model as a web service. The biggest advantage of using Streamlit is that it allows you to use HTML code within the application Python file. It doesn't essentially require separate templates and CSS formatting for the front-end UI. However, it is suggested that you create separate folders for templates and style guides for a more complex application. To install Streamlit, we can simply use `pip` to install Streamlit in our terminal.

We are going to use the same dataset that we used when building the Flask app model. The only content that is going to change is in the `app.py` file. The first set of commands is to import the required libraries such as `joblib` and `streamlit`.

```
[In]: import pandas as pd
[In]: import numpy as np
[In]: import joblib
[In]: import streamlit
```

In the next step, we import the trained linear regression model to be able to predict on the test data.

```
[In]: model=open("linear_regression_model.pkl","rb")
[In]: lr_model=joblib.load(model)
```

The next step is to define a function to make predictions using the trained model. We pass the five input parameters in the function and do a bit of reshaping and data casting to ensure consistency for predictions. Then we create a variable to save the model predictions result and return it to the user.

```
[In]: def lr_prediction(var_1,var_2,var_3,var_4,var_5):
      pred_arr=np.array([var_1,var_2,var_3,var_4,var_5])
```

```

preds=pred_arr.reshape(1,-1)
preds=preds.astype(int)
model_prediction=lr_model.predict(preds)
return model_prediction

```

In the next step, we create the most important function. We accept the user input from the browser and render the model's final predictions on the web page. We can name this function anything. For example, I have used `run` (since it does the same thing as Flask's `app.run`). In this function, we include the front-end code as well such as defining the title, theme, color, background, etc. For simplicity purposes, I have kept it basic, but this can have multiple levels of enhancements. For more details, you can visit the Streamlit website. Next, we create five input variables to accept the user input values from the browser. This is done using Streamlit's `text_input` capability. The final part contains the model prediction, which gets the input from our `lr_prediction` function defined earlier and gets rendered in the browser through `streamlit.button`.

```

[In]: def run():
        streamlit.title("Linear Regression Model")
        html_temp="""
        """
        streamlit.markdown(html_temp)
var_1=streamlit.text_input("Variable 1")
var_2=streamlit.text_input("Variable 2")
var_3=streamlit.text_input("Variable 3")
var_4=streamlit.text_input("Variable 4")
var_5=streamlit.text_input("Variable 5")

```

```

prediction=""

if streamlit.button("Predict"):
    prediction=lr_prediction(var_1,var_2,var_3,
                             var_4,var_5)
streamlit.success("The prediction by Model : {}".
format(prediction))

```

Now that we have all the steps mentioned in the application file, we can call the main function (run in our case) and use the `streamlit run` command to run the app.

```

[In]: if __name__=='__main__':
      run()
[In]: streamlit run app.py

```

Once we run the previous command, we will soon see the app up and running on port 8501, as shown in Figure 3-6. We can simply click the link and access the app.

You can now view your Streamlit app in your browser.

Local URL: **http://localhost:8501**
 Network URL: **http://192.168.1.4:8501**

Figure 3-6. Access running app

Once we are at `http://localhost:8501/`, we will see the screen shown in Figure 3-7.

Linear Regression Model

Variable 1

Variable 2

Variable 3

Variable 4

Variable 5

Predict

The prediction by Model is

Figure 3-7. *User input page*

As you can see, we have nothing fancy here in the UI, but it serves the overall purpose for us to be able to interact with the model/app behind the scenes. It is similar to what we had built using Flask, but it needed much less HTML and CSS code. We can now go ahead and fill in the values, as shown in Figure 3-8, and get the model prediction. For comparison sake, we fill in the same values that we filled in the Flask-based app.

Linear Regression Model

Variable 1
34728

Variable 2
32123

Variable 3
213123

Variable 4
12312

Variable 5
23412

Predict

The prediction by Model is

Figure 3-8. *Providing input values for the model*

After filling in the values, we need to click the Predict button to fetch the model prediction result and *voilà*—it’s the same number that we got in the Flask-based app, as shown in Figure 3-9.

Linear Regression Model

Variable 1

34728

Variable 2

32123

Variable 3

213123

Variable 4

12312

Variable 5

23412

Predict

The prediction by Model is [3608.45201764]

Figure 3-9. *Model prediction*

Now that we have seen how to deploy a traditional machine learning model using Flask and Streamlit, we can move on to the last topic of this chapter, which focuses on deploying a deep learning model (LSTM) as a web service.

Deploying a Deep Learning Model

In the previous sections of the chapter, we covered the process to build and deploy a machine learning model (linear regression) using two different frameworks.

- Flask
- Streamlit

In this section, we will see how to build a deep learning model and deploy it using Streamlit. In particular, we build an LSTM-based neural network to predict the sentiment of a given review. LSTM/RNNs are known to be a powerful way to build sequential models and capture the order of the input for predictions. We will use an open source reviews dataset and train an LSTM model to predict the sentiment of the review. We then serve this model as a web service using Streamlit. Again, in the previous cases, the focus is not to train a perfect model, rather a decent model, and deploy it. The underlying principles remain the same as previously except for some of the additional components such as text processing and a tokenizer to handle the incoming data (user review). We are making use of Streamlit to avoid a whole lot of template, HTML, and CSS stuff and make it easy to deploy the deep learning model.

Training the LSTM Model

For those of you who are new to RNN/LSTMs, I recommend you read more about them to understand how they work and when they can be used for predictions. The first step is to import all the required libraries.

```
[In]: import numpy as np
[In]: import pandas as pd
[In]: from tensorflow.keras.models import Sequential
[In]: from tensorflow.keras.layers import LSTM, Embedding
[In]: from tensorflow.keras.layers import Dense
[In]: from tensorflow.keras.preprocessing.text import Tokenizer
[In]: from tensorflow.keras.preprocessing.sequence import pad_
sequences
[In]: from sklearn.model_selection import train_test_split
[In]: from keras.utils.np_utils import to_categorical
[In]: import re
[In]: import pickle
```


The next step is to load the data and check the size of the data.

```
[In]: df = pd.read_csv('reviews_dataset.tsv.zip', header=0,
delimiter="\t", quoting=3)
```

```
[In]: df = df[['review', 'sentiment']]
```

As we can see, we have 25,000 records in our dataset and two equal categories of sentiments.

```
[In]: df.shape
```

```
[Out]: (25000, 2)
```

```
[In]: df.sentiment.value_counts()
```

```
[Out]:
```

```
1    12500
```

```
0    12500
```

Next, we apply bit of text preprocessing to clean the reviews using regular expressions.

```
[In]: df['review'] = df['review'].apply(lambda x: x.lower())
```

```
[In]: df['review'] = df['review'].apply((lambda x: re.sub('[^a-
zA-z0-9\s]', '', x)))
```

We restrict the number of features to 1,000 and tokenize the reviews and add padding to make each review the same size.

```
[In]: max_features = 1000
```

```
[In]: tokenizer = Tokenizer(num_words=max_features, split=' ')
```

```
[In]: tokenizer.fit_on_texts(df['review'].values)
```

```
[In]: X = tokenizer.texts_to_sequences(df['review'].values)
```

```
[In]: X = pad_sequences(X)
```

```
[In]: X.shape
```

```
(25000, 1473)
```

We then keep the embedding layer size as 50.

```
[In]: embed_dim = 50

[In]: model = Sequential()
[In]: model.add(Embedding(max_features, embed_dim,
    input_length = X.shape[1]))
[In]: model.add(LSTM(10))
[In]: model.add(Dense(2,activation='softmax'))
[In]: model.compile(loss = 'categorical_crossentropy',
    optimizer='adam',metrics = ['accuracy'])
[In]: print(model.summary())
[Out]:
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 1473, 50)	50000
lstm (LSTM)	(None, 10)	2440
dense (Dense)	(None, 2)	22
Total params: 52,462		
Trainable params: 52,462		
Non-trainable params: 0		

```
[In]: y = pd.get_dummies(df['sentiment']).values
[In]: X_train, X_test, y_train, y_test = train_test_split
    (X,y, test_size = 0.25, random_state = 99)
[In]: print(X_train.shape,y_train.shape)
[In]: print(X_test.shape,y_test.shape)

[Out]:
(18750, 1473) (18750, 2)
(6250, 1473) (6250, 2)
```

```
[In]: model.fit(X_train, y_train, epochs = 5, verbose = 1)
```

```
Train on 18750 samples
Epoch 1/5
18750/18750 [=====] - 263s 14ms/sample - loss: 0.4837 - accuracy: 0.7610
Epoch 2/5
18750/18750 [=====] - 278s 15ms/sample - loss: 0.3485 - accuracy: 0.8545
Epoch 3/5
18750/18750 [=====] - 273s 15ms/sample - loss: 0.3279 - accuracy: 0.8623
Epoch 4/5
18750/18750 [=====] - 274s 15ms/sample - loss: 0.3077 - accuracy: 0.8709
Epoch 5/5
18750/18750 [=====] - 270s 14ms/sample - loss: 0.2922 - accuracy: 0.8765
```

Now that we have trained the LSTM model, let's try to pass a test review to see the predictions by the model.

```
[In]: test = ['Movie was pathetic']
[In]: test = tokenizer.texts_to_sequences(test)
[In]: test = pad_sequences(test, maxlen=X.shape[1],
    dtype='int32', value=0)
[In]: print(test.shape)
[In]: sentiment = model.predict(test)[0]
    if(np.argmax(sentiment) == 0):
        print("Negative")
    elif (np.argmax(sentiment) == 1):
        print("Positive")
[Out]: Negative
```

As we can see, the model is able to predict well on the test review. The next step is to save the model and tokenizer using pickle and load it later for making predictions on user input reviews.

```
[In]: with open('tokenizer.pickle', 'wb') as tk:
    pickle.dump(tokenizer, tk, protocol=pickle.HIGHEST_
        PROTOCOL)

[In]: model_json = model.to_json()
    with open("model.json", "w") as js:
js.write(model_json)
```

```
[In]: model.save_weights("model.h5")
```

Now that we have saved the trained model and tokenizer, we can create the application script similar to the earlier `app.py` script.

```
[In]: import os
[In]: import numpy as np
[In]: import pandas as pd
[In]: import pickle
[In]: import tensorflow
[In]: from tensorflow.keras.preprocessing.text import Tokenizer
[In]: from tensorflow.keras.preprocessing.sequence import pad_
    sequences
[In]: import tensorflow.keras.models
[In]: from tensorflow.keras.models import model_from_json
[In]: import streamlit
[In]: import re
[In]: os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
```

The step after importing the required libraries is to load the tokenizer and deep learning model.

```
[In]: with open('tokenizer.pickle', 'rb') as tk:
    tokenizer = pickle.load(tk)

[In]: json_file = open('model.json', 'r')
[In]: loaded_model_json = json_file.read()
[In]: json_file.close()
[In]: lstm_model = model_from_json(loaded_model_json)

[In]: lstm_model.load_weights("model.h5")
```

Next, we create a helper function to clean the input review, tokenize it, and pad the sequence. Once it's converted into numerical form, we will use the loaded LSTM model to make the sentiment prediction.

```
[In]: def sentiment_prediction(review):
    sentiment=[]
    input_review = [review]
    input_review = [x.lower() for x in input_review]
    input_review = [re.sub('[^a-zA-Z0-9\s]','',x) for x in input_
review]

    input_feature = tokenizer.texts_to_sequences(input_review)
    input_feature = pad_sequences(input_feature,1473,
padding='pre')
    sentiment = lstm_model.predict(input_feature)[0]

    if(np.argmax(sentiment) == 0):
pred="Negative"
    else:
pred= "Positive"

    return pred
```

At the end, we create the run function to load the HTML page and accept the user input using Streamlit functionality (similar to the earlier model deployment). The only difference is that instead of loading multiple inputs, this time we load just a single review. We then pass this review to the sentiment prediction function created earlier.

```
[In]: def run():
streamlit.title("Sentiment Analysis - LSTM Model")
html_temp="""

"""

streamlit.markdown(html_temp)
    review=streamlit.text_input("Enter the Review ")
    prediction=""
```

```

    if streamlit.button("Predict Sentiment"):
        prediction=sentiment_prediction(review)
    streamlit.success("The sentiment predicted by Model : {}".
format(prediction))

```

```

[In]: if __name__=='__main__':
    run()

```

Once we run the `app.py` file using the `streamlit run` command in the terminal, we can see the web service is running on localhost port 8501, as shown in Figure 3-10.

```

[In]: streamlit run app.py

```

You can now view your Streamlit app in your browser.

Local URL: **http://localhost:8501**
 Network URL: **http://192.168.1.4:8501**

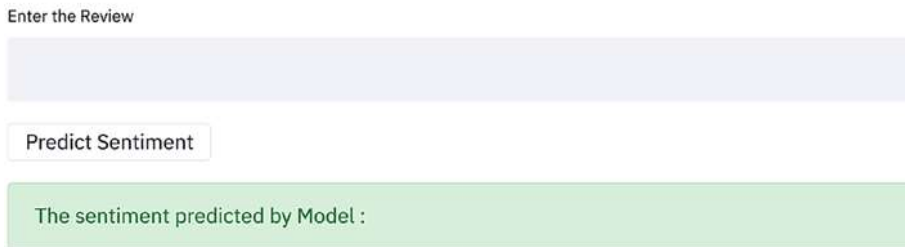
Figure 3-10. *Accessing the ML app*

We can access the ML service on port 8501 and can see the ML app running successfully. The page contains three things.

- Placeholder to provide a user review
- A Predict Sentiment button (to make a prediction using the model)
- The final result by the model

Let's provide a review in the input box and check the model predictions, as shown in Figure 3-11. First, we provide a positive review and click the Predict Sentiment button, as shown in Figure 3-12.

Sentiment Analysis - LSTM Model



Enter the Review

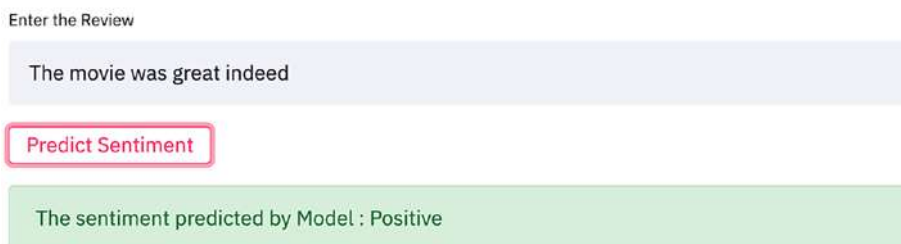
Predict Sentiment

The sentiment predicted by Model :

This figure shows a web interface for sentiment analysis. It has a title 'Sentiment Analysis - LSTM Model'. Below the title is a text input field with the placeholder 'Enter the Review'. Below the input field is a button labeled 'Predict Sentiment'. Below the button is a green box with the text 'The sentiment predicted by Model :', indicating the output area.

Figure 3-11. *User input page*

Sentiment Analysis - LSTM Model



Enter the Review

The movie was great indeed

Predict Sentiment

The sentiment predicted by Model : Positive

This figure shows the same web interface as Figure 3-11, but with a review entered. The text input field now contains 'The movie was great indeed'. The 'Predict Sentiment' button is highlighted with a red border. The green output box now displays 'The sentiment predicted by Model : Positive'.

Figure 3-12. *Positive review prediction*

As we can observe, the outcome of the model prediction seems correct as it also predicts it to be a positive review. We try with another review—negative this time—and test for the model prediction, as shown in Figure 3-13. For the second review as well, we get a correct prediction by the model. We can replace the existing model with a much more powerful, well-trained, and optimized model to have better predictions.

Sentiment Analysis - LSTM Model

Enter the Review

The movie very bad, pathetic story and acting

Predict Sentiment

The sentiment predicted by Model : Negative

Figure 3-13. *Negative review prediction*

Conclusion

In this chapter, we went over the fundamentals of Flask and its different components. We also saw the process of building and deploying a machine learning model using Flask. In the end, we explored another platform to deploy ML models called Streamlit and its advantages over Flask.