Data Glacier

Data Science Intern at Data Glacier

Week 5: Cloud and API Deployment

Name: Yash Jayesh Doshi

Batch Code: LISUM19

Date: 01st April 2022

Submitted to: Data Glacier

Table of Contents:

Heading	Page number
Introduction	3
Data Information	3
Building a model in Jupyter notebook	4
Building a Flask Web Application	9
Checking the Flask Web Application	17
API Testing	19
API Deployment	19
Deployment on Cloud	20

1. Introduction

In this project, I try to predict the sentiment of a movie review with the help of NLP and ML. The overall workflow of the project is shown in the figure below.

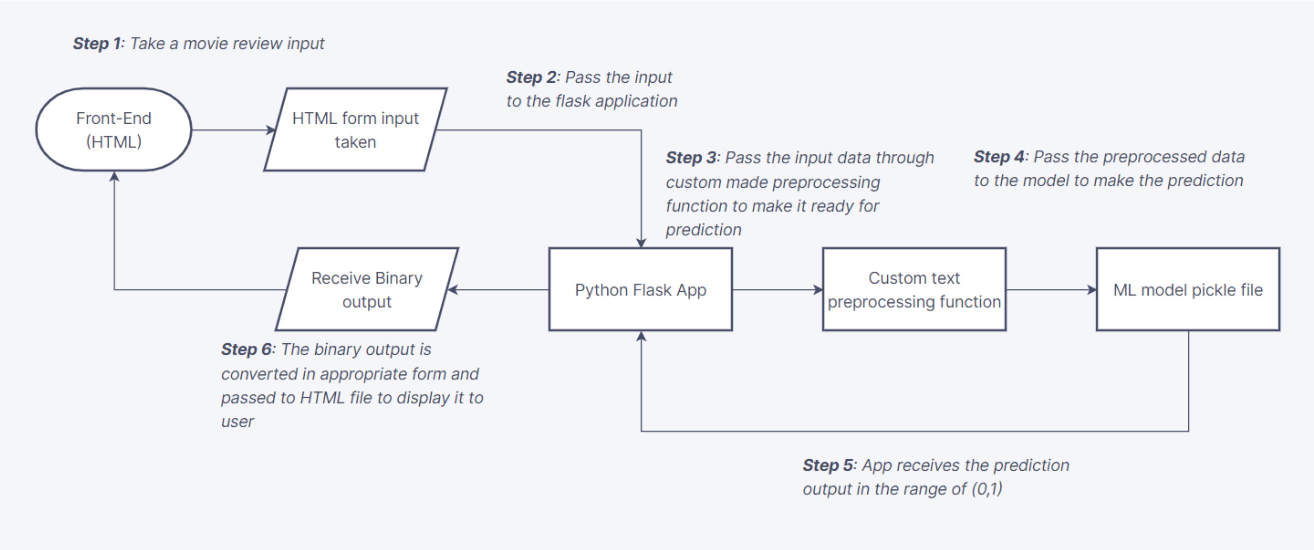

Figure 1.1: Application Workflow

2. Data Information

The data used for this project is IMDb movie reviews dataset (Link)

Source credit:

```
      @InProceedings{maas-EtAl:2011:ACL-HLT2011,
  author    = {Maas, Andrew L.  and  Daly, Raymond E.  and  Pham, Peter T.  and  Huang, Dan  and  Ng, Andrew Y.  and  Potts, Christopher},
  title     = {Learning Word Vectors for Sentiment Analysis},
  booktitle = {Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies},
  month     = {June},
  year      = {2011},
  address   = {Portland, Oregon, USA},
  publisher = {Association for Computational Linguistics},
  pages     = {142--150},
  url       = {http://www.aclweb.org/anthology/P11-1015}
}
```

This dataset contains movie reviews along with their associated binary sentiment polarity labels. It is intended to serve as a benchmark for sentiment classification.

The core dataset contains 50,000 reviews split evenly into 25,000 train and 25,000 test sets. The overall distribution of labels is balanced (25,000 pos and 25,000 neg). In the entire collection, no more than 30 reviews are allowed for any given movie because reviews for the same movie tend to have correlated ratings. Further, the train and test sets contain a disjoint set of movies, so no significant performance is obtained by memorizing movie-unique terms and their associated with observed labels. In the labeled train/test sets, a negative review has a score <= 4 out of 10, and a positive review has a score >= 7 out of 10. Thus, reviews with more neutral ratings are not included in the train/test sets.

There are two top-level directories [train/, test/] corresponding to the training and test sets. Each contains [pos/, neg/] directories for the reviews with binary labels positive and negative. Within these directories, reviews are stored in text files named following the convention [[id]_[rating].txt] where [id] is a unique id and [rating] is the star rating for that review on a 1-10 scale. For example, the file [test/pos/200_8.txt] is the text for a positive-labeled test set example with unique id 200 and star rating 8/10 from IMDb.

3. Building a Model in Jupyter Notebook

3.1 Importing the Required Libraries and Dataset

```
import numpy as np
import pandas as pd
import warnings
import os
import pickle

from sklearn.model_selection import train_test_split
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.pipeline import make_pipeline
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import classification_report

warnings.filterwarnings("ignore")
nltk.download('stopwords')
nltk.download('wordnet')
```

Figure 3.1: Importing the required libraries

Loading the data

```
# Load the reviews and labels
# 1 for positive reviews and 0 for negative reviews
reviews = []
labels = []
path="aclImdb/train"
for label_type in ["pos", "neg"]:
    dir_path = os.path.join(path, label_type)
    for filename in os.listdir(dir_path):
        if filename.endswith(".txt"):
            with open(os.path.join(dir_path, filename), "r", encoding="utf-8") as f:
                review = f.read()
                reviews.append(review)
                labels.append(1 if label_type == "pos" else 0)
```

Figure 3.2: Loading the data

3.2 Preprocessing the data

```
# Define a function to perform text preprocessing
def preprocess_text(text):
    # Remove HTML tags
    cleaned_text = re.sub('<[^>]*>', '', text)

    # Remove punctuation
    cleaned_text = re.sub(r'[^\w\s]', '', cleaned_text)

    # Convert to lowercase
    cleaned_text = cleaned_text.lower()

    #Lemmatize and remove stopwords
    lemmatizer=WordNetLemmatizer()
    stop_words = set(stopwords.words('english'))
    cleaned_text=' '.join(lemmatizer.lemmatize(word) for word in cleaned_text.split()
                                                     if word not in stop_words)

    return cleaned_text

# Preprocess the movie reviews
cleaned_reviews = []
for review in reviews:
    cleaned_review = preprocess_text(review)
    cleaned_reviews.append(cleaned_review)
```

Figure 3.3: Preprocessing the reviews

3.3 Splitting the dataset into train, validation and test set

```
# Split the dataset into training, validation, and testing sets
train_reviews, test_reviews, train_labels, test_labels = train_test_split(cleaned_reviews, labels, test_size=0.15,
                                                                          random_state=42, shuffle=True, stratify=labels)
train_reviews, val_reviews, train_labels, val_labels = train_test_split(train_reviews, train_labels, test_size=0.15,
                                                                        random_state=42, shuffle=True, stratify=train_labels)
```

Figure 3.4: Splitting the dataset

3.4 Vectorizing the data and finding the best ML model

```
# Making a pipeline for vectorizing and transforming the reviews
pipeline = make_pipeline(CountVectorizer(),TfidfTransformer())

X_train = pipeline.fit_transform(train_reviews)
X_val = pipeline.transform(val_reviews)
X_test = pipeline.transform(test_reviews)
```

```
# Defining the models that will be tested for our analysis
models = {
    "MultinomialNB": MultinomialNB(),
    "Logistic Regression": LogisticRegression(),
    "SVM": SVC(kernel="linear"),
    "Random Forest": RandomForestClassifier(),
    "XGBoost Classifier": XGBClassifier()
}
```

Figure 3.5: Transforming the data and defining models dictionary

```python
from sklearn.model_selection import GridSearchCV
import time

model_list=[]

# Define the parameter grid for each model
nb_param_grid = {'alpha': [0.1, 0.01, 0.001, 0.0001]}

lr_param_grid = {'C': [0.1, 1, 10, 100],
                 'penalty': ['l1', 'l2']}

svm_param_grid = {'C': [0.1, 1, 10, 100]}

rf_param_grid = {'n_estimators': [50, 100, 200, 300],
                 'max_depth': [None, 5, 10, 20],
                 'min_samples_split': [2, 5, 10]}

xgb_param_grid = {'n_estimators': [50, 100, 200, 300],
                  'max_depth': [3, 5, 10],
                  'learning_rate': [0.1, 0.01, 0.001]}

# Define the parameter grids for all models in a dictionary
param_grids = {'MultinomialNB': nb_param_grid,
               'Logistic Regression': lr_param_grid,
               'SVM': svm_param_grid,
               'Random Forest': rf_param_grid,
               'XGBoost Classifier': xgb_param_grid}
```

Figure 3.6: Importing required libraries and defining hyper-parameters dictionary

```python
# Testing all the models and appending the statistics to model_list
for model_name,param in param_grids.items():

    start = time.time()
    algo = models[model_name]
    grid_search = GridSearchCV(estimator=algo,param_grid=param,cv=5,n_jobs=-1,scoring='accuracy')
    grid_search.fit(X_train,train_labels)
    best_para = grid_search.best_params_
    algo.set_params(**best_para)
    algo.fit(X_train,train_labels)
    y_pred = algo.predict(X_val)
    end = time.time()

    print(f"{model_name}")
    print(f"Time taken: {end-start}") #Time is in seconds
    model_list.append([model_name,accuracy_score(val_labels,y_pred),precision_score(val_labels,y_pred),
                       recall_score(val_labels,y_pred),f1_score(val_labels,y_pred),best_para])
    print("----------------------------------------------------")
```

```
MultinomialNB
Time taken: 6.968690395355225
----------------------------------------------------
Logistic Regression
Time taken: 20.587644577026367
----------------------------------------------------
SVM
Time taken: 1976.6532769203186
----------------------------------------------------
Random Forest
Time taken: 4465.19069314003
----------------------------------------------------
XGBoost Classifier
Time taken: 5181.1491086483
----------------------------------------------------
```

Figure 3.7: Finding the best model and best hyper-parameters

```python
models_df = pd.DataFrame(data=model_list,columns=['Model name','Accuracy','Precision','Recall','f1-score','Best Parameters'])
models_df
```

	Model name	Accuracy	Precision	Recall	f1-score	Best Parameters
0	MultinomialNB	0.860100	0.865140	0.853199	0.859128	{'alpha': 0.1}
1	Logistic Regression	0.896801	0.888275	0.907779	0.897921	{'C': 10, 'penalty': 'l2'}
2	SVM	0.898996	0.890663	0.909661	0.900062	{'C': 1}
3	Random Forest	0.861669	0.868842	0.851945	0.860310	{'max_depth': None, 'min_samples_split': 5, 'n...
4	XGBoost Classifier	0.863237	0.856527	0.872647	0.864512	{'learning_rate': 0.1, 'max_depth': 10, 'n_est...

Figure 3.8: Viewing the results in a dataframe

3.5 Choosing the best model, training it and saving the model and pipeline

Choosing Logistic Regression for our analysis

```
model = LogisticRegression(C=10,penalty='l2')
model.fit(X_train,train_labels)
prediction = model.predict(X_test)
```

```
print(classification_report(prediction,test_labels))
```

```
              precision    recall  f1-score   support

           0       0.86      0.89      0.88      1825
           1       0.89      0.87      0.88      1925

    accuracy                           0.88      3750
   macro avg       0.88      0.88      0.88      3750
weighted avg       0.88      0.88      0.88      3750
```

```
with open('ml_model.pickle', 'wb') as f:
    pickle.dump(model, f)
```

```
with open('preprocessor.pickle', 'wb') as f:
    pickle.dump(pipeline, f)
```

Figure 3.9: Fitting the data to the best model and saving the model

4. Building the Flask Web Application

Now that we have trained and saved the model in a pickle file, we will use it to predict the sentiment of a movie review entered by a user in our Web Application.

4.1 Building utils.py

We will define all the preprocessing operations in a function inside the file utils.py, which we can then use to preprocess the movie review that we get from the user of our application.

utils.py > text_preprocessor

```python
import numpy as np
import re
import pickle
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

file_path = 'preprocessor.pickle'
with open(file_path,'rb') as f:
    preprocessor_pipeline = pickle.load(f)

def text_preprocessor(text):
    # Remove HTML tags
    preprocessed_text = re.sub('<[^>]*>', '', text)

    # Remove punctuation
    preprocessed_text = re.sub(r'[^\w\s]', '', preprocessed_text)

    # Convert to lowercase
    preprocessed_text = preprocessed_text.lower()

    #Lemmatize and remove stopwords
    lemmatizer=WordNetLemmatizer()
    stop_words = set(stopwords.words('english'))
    preprocessed_text=' '.join(lemmatizer.lemmatize(word) for word in preprocessed_text.split()
                                                          if word not in stop_words)

    preprocessed_text = preprocessor_pipeline.transform([preprocessed_text])

    return preprocessed_text
```

Figure 4.1: utils.py

4.2 Building app.py

Now, we will build our Flask application in app.py. We will use the default page to take the input from the user, and display the input and the prediction in a separate web page. We will locally store all the inputs the user provides so user can view the predictions of multiple movie reviews.

Additionally, we will define a separate URL to test the app on an API software.

The entire code for app.py can be found below.

```python
app.py > ...
from flask import Flask, render_template, request, Response, url_for, jsonify
import tensorflow as tf
import pickle

from utils import text_preprocessor

file_path = 'ml_model.pickle'
with open(file_path,'rb') as f:
    model = pickle.load(f)

app = Flask(__name__)

# Defining route for home page
@app.route('/')
def index():
    return render_template("index.html")

# Defining route for api
@app.route('/predict_api',methods=['GET','POST'])
def predict_api():
    if request.method == 'POST':

        data = request.json['data']
        preprocessed_data = text_preprocessor(data)
        output = model.predict(preprocessed_data)[0][0]

        return jsonify(output)
```

Figure 4.2: app.py (part-1)

```python
@app.route('/predict',methods=['GET','POST'])
def predict():
    if request.method == 'POST':
        review=request.form['review']
        preprocessed_data = text_preprocessor(review)
        output = model.predict(preprocessed_data)
        if output>0.5:
            sentiment="Positive"
        else:
            sentiment="Negative"
        predictions.append({'review': review, 'sentiment': sentiment})
        return render_template('predictions.html', predictions=predictions)
    return render_template("index.html")

@app.route('/predictions')
def predictions():
    # Render the predictions page with the list of predictions
    return render_template('predictions.html', predictions=predictions)

if __name__=="__main__":
    predictions=[]
    app.run()
```

Figure 4.3: app.py (part-2)

4.3 Building the HTML and CSS files for the application

Our model includes two HTML files, namely, index.html and predictions.html, and one CSS file to style the two HTML files, namely, style.css

The code for both the HTML files can be found below.

```
templates > index.html > html
<!DOCTYPE html>

<html>
  <head>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css') }}">
    <title>Movie Review Sentiment Analysis</title>
  </head>
  <body>
    <h1>Movie Review Sentiment Analysis</h1>
    <form action="/predict" method="post">
      <label for="review">Enter your movie review:</label>
      <br>
      <textarea id="review" name="review" rows="10" cols="50"></textarea>
      <br>
      <button type="submit">Submit</button>
    </form>
  </body>
</html>
```

Figure 4.4: index.html

```
templates > predictions.html > html
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css') }}">
    <title>Movie Review Sentiment Analysis Predictions</title>
  </head>
  <body>
    <h1>Movie Review Sentiment Analysis Predictions</h1>
    <table>
      <thead>
        <tr>
          <th>Review</th>
          <th>Sentiment</th>
        </tr>
      </thead>
      <tbody>
        {% for prediction in predictions %}
        <tr>
          <td class="review">{{ prediction.review }}</td>
          <td class="{{ 'positive' if prediction.sentiment=='Positive' else 'negative'}}">{{ prediction.sentiment }}</td>
        </tr>
        {% endfor %}
      </tbody>
    </table>
    <a class="New_predictions" href="/">Make a new prediction</a>
  </body>
</html>
```

Figure 4.5: predictions.html

The code for the CSS file can be found below.

```css
/* Styles for the index page */
body {
  background-color: #f9f9f9;
  font-family: 'Helvetica Neue', sans-serif;
  color: #333;
}

h1 {
  text-align: center;
  font-size: 3rem;
  margin-top: 3rem;
}

form {
  margin: 2rem auto;
  max-width: 800px;
  padding: 2rem;
  background-color: #fff;
  border-radius: 10px;
  box-shadow: 0 0 20px rgba(0, 0, 0, 0.2);
}

label {
  display: block;
  font-size: 1.2rem;
  font-weight: bold;
  margin-bottom: 1rem;
}

```

Figure 4.6: style.css (part-1)

```css
textarea {
  display: block;
  width: 100%;
  font-size: 1.2rem;
  padding: 0.5rem;
  border-radius: 5px;
  border: 1px solid #ccc;
  resize: none;
  height: 200px;
}

button {
  display: block;
  margin: 1rem auto;
  padding: 0.5rem 1rem;
  font-size: 1.2rem;
  background-color: #007bff;
  color: #fff;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

/* Styles for the predictions page */
table {
  margin: 2rem auto;
  max-width: 1100 px;
  border-collapse: collapse;
}

thead {
  background-color: #007bff;
  color: #fff;
}

```

Figure 4.7: style.css (part-2)

```css
th, td {
  padding: 0.5rem;
  text-align: center;
  border: 1px solid #ccc;
}

th {
  font-weight: bold;
}

td.review {
  padding-top: 20px;
  padding-bottom: 20px;
  padding-right: 10px;
  text-align: left;
}

tbody tr:nth-child(even) {
  background-color: #f2f2f2;
}

a.New_predictions {
  display: flex;
  justify-content: center;
  margin: 1rem auto;
  padding: 0.5rem 1rem;
  font-size: 1.2rem;
  background-color: #007bff;
  color: #fff;
  border: none;
  border-radius: 5px;
  text-align: center;
  text-decoration: none;
  cursor: pointer;
}

```

Figure 4.8: style.css (part-3)

```css
a.New_Predictions:hover {
  background-color: #0056b3;
}

.positive {
  color: green;
  font-weight: bold;
}

.negative {
  color: red;
  font-weight: bold;
}

```

Figure 4.9: style.css (part-4)

4.4 Building the Flask Application API using flask-restful library

```python
from flask import Flask, request, jsonify, redirect
from flask_restful import Resource, Api
import pickle

from utils import text_preprocessor

file_path = 'ml_model.pickle'
with open(file_path, 'rb') as f:
    model = pickle.load(f)

app = Flask(__name__)
api = Api(app)

@app.route("/")
def base_route():
    return redirect("/predict_api")

class PredictAPI(Resource):

    def get(self):
        data = """Titanic is just one of its kind. Never have I seen a movie so well put, well-directed and well-executed.
            Eventhough the climax was mediocre, it was all in all a good watch."""
        preprocessed_data = text_preprocessor(data)
        output = model.predict(preprocessed_data)
        sentiment = "Positive" if output > 0.5 else "Negative"

        response = {"data": data, "sentiment": sentiment}
        return response, 200


api.add_resource(PredictAPI, '/predict_api')

if __name__ == "__main__":
    app.run()
```

5. Checking the Flask Web Application

The images below depict how the Web Application works.

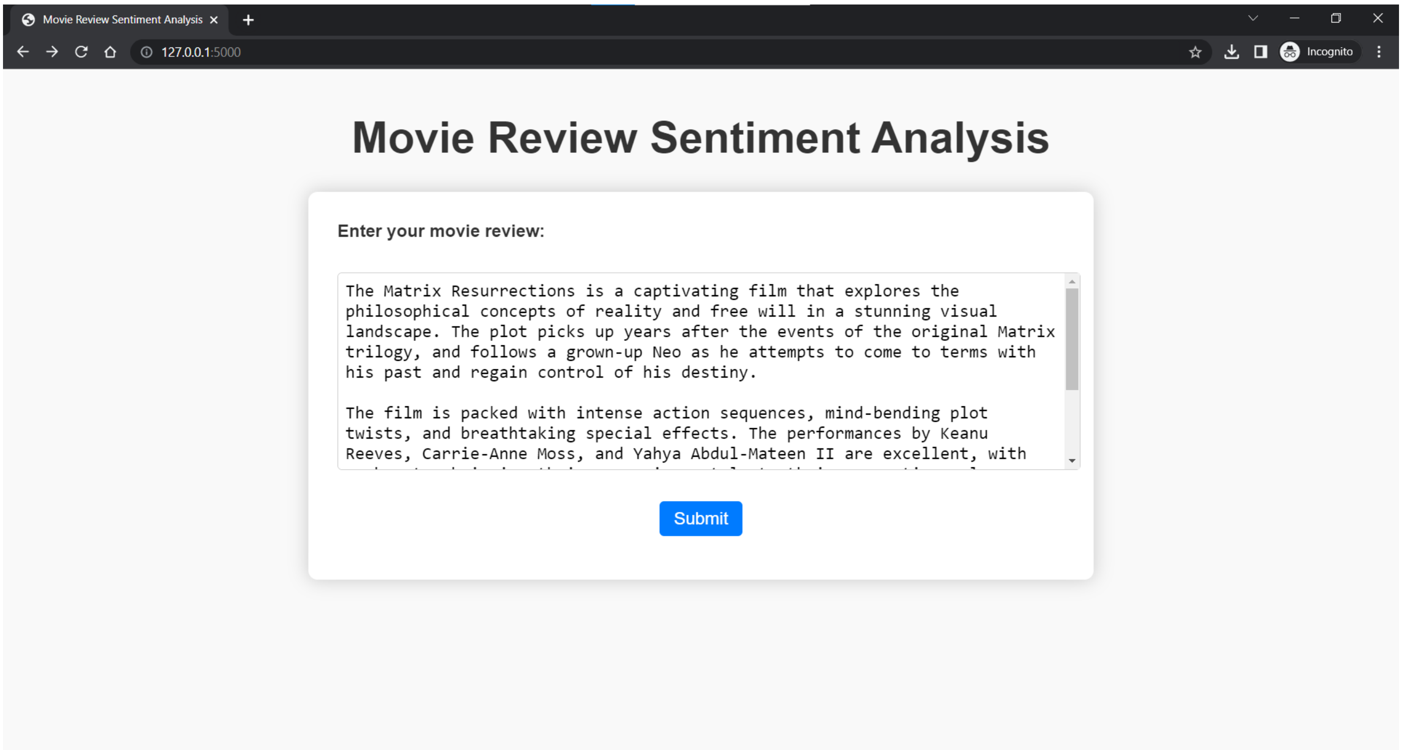

Figure 5.1: Entering movie review and submitting it

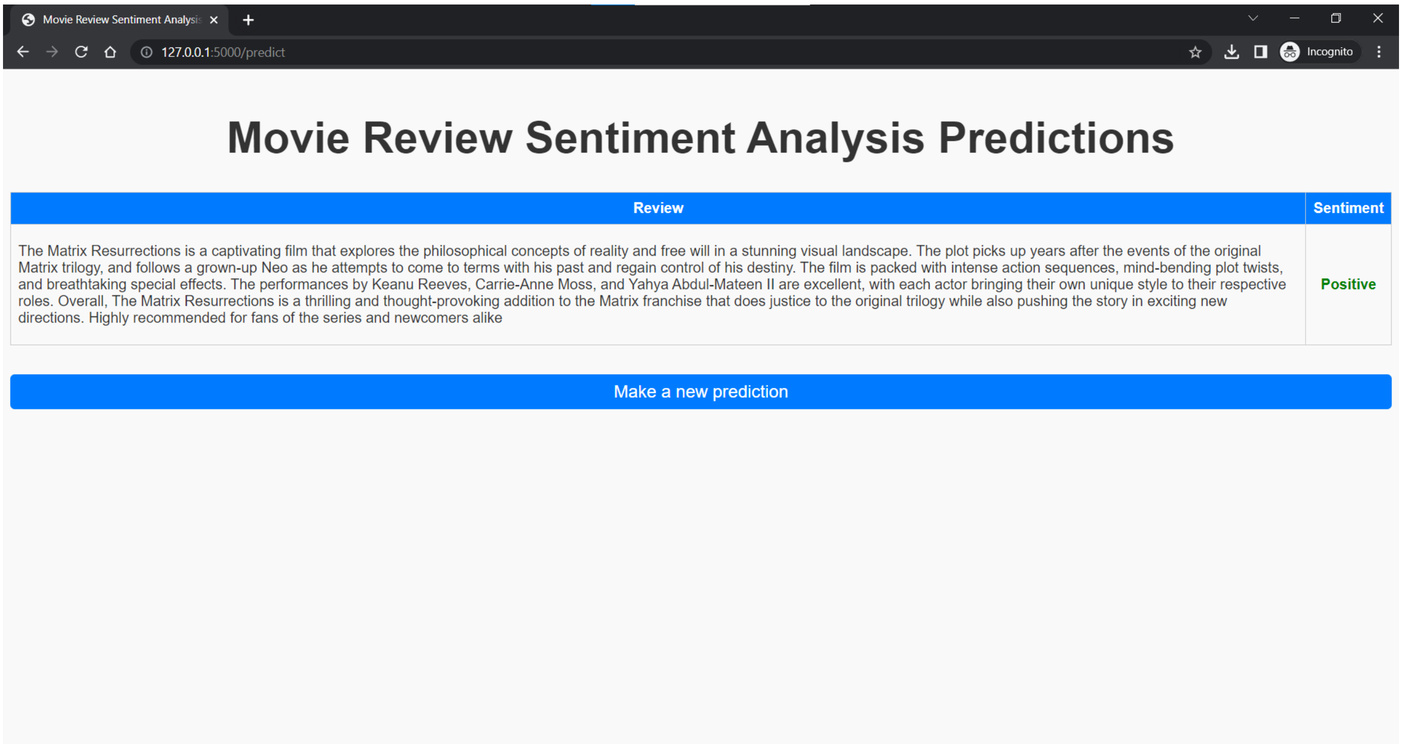

Figure 5.2: Displaying the input and output and clicking 'Make a New Prediction'

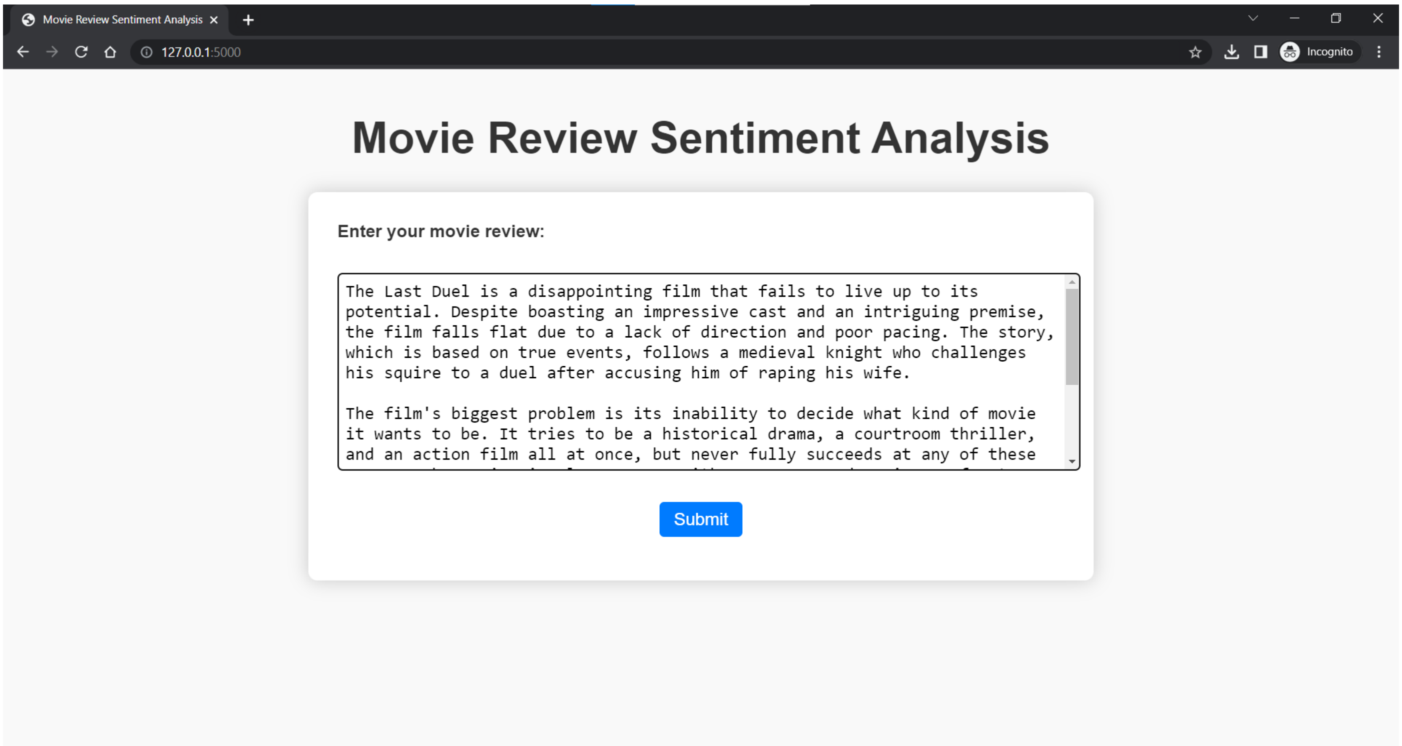

Figure 5.3: Entering second movie review

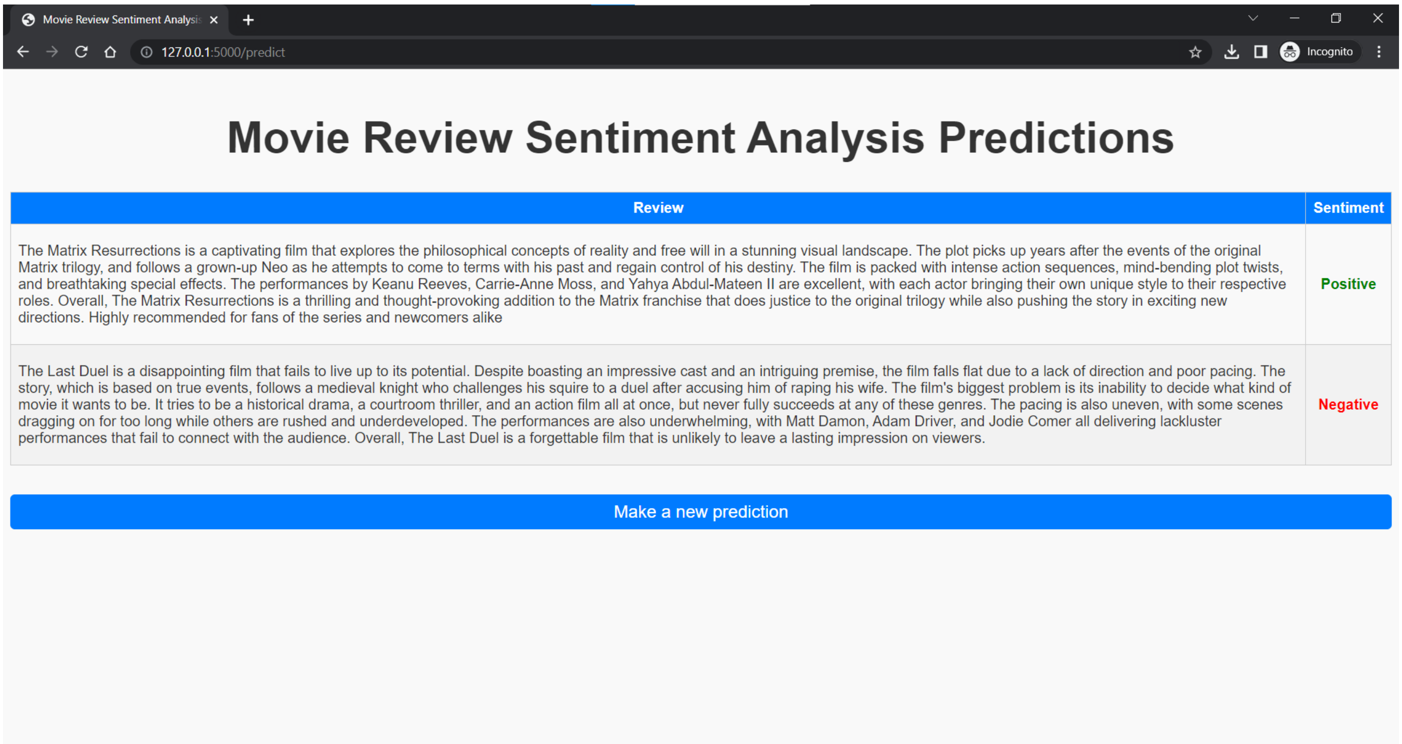

Figure 5.4: Displaying input and corresponding output for all the reviews

6. API Testing

We can also test our flask app through an API tester. We will use POSTMAN for our purpose which will use the ‘/predict_api’ app-route to make the predictions of the movie review which we will provide. The below image depicts the implementation.

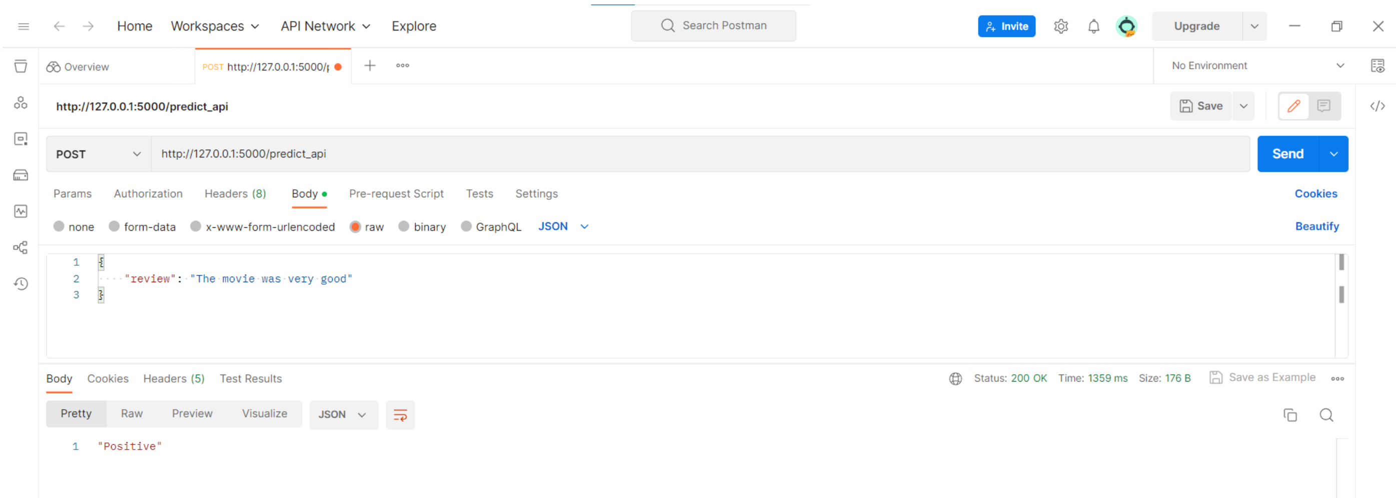

Figure 6.1: API Testing on POSTMAN

7. Deployment of Api on Google Cloud

We first create the necessary files to successfully upload the API to Google Cloud Platform (GCP). These include app.yaml and requirements.txt. Additionally, we need to rename the API file to main.py to ensure error-free implementation.

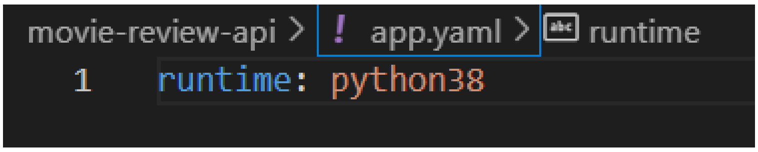

Figure 7.1: app.yaml

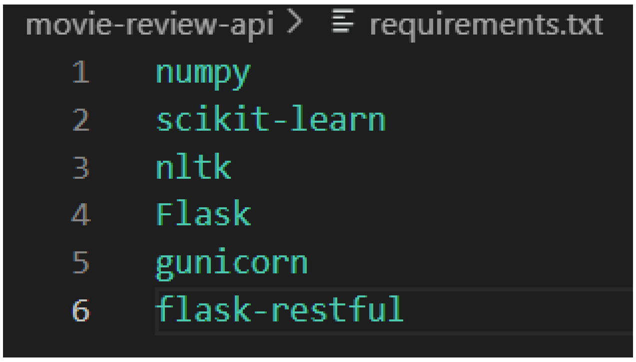

Figure 7.2: requirements.txt

Deploying the API to the cloud involves running the following commands after authenticating the user and selecting the project.

1. ***gcloud app create:*** This command creates the app infrastructure in the selected project in the Google Cloud.
2. ***gcloud app deploy:*** This command deploys the api app and the related files to the project folder on the cloud.
3. ***gcloud app browse:*** This command finally runs the deployed API on the Google Cloud Platform.

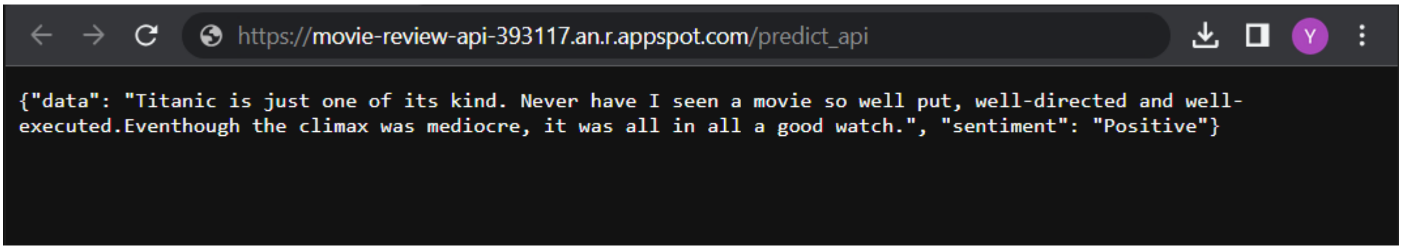

Figure 7.3: API Deployment to Google Cloud Platform

8. Deployment of App on Amazon Cloud

Now, we will deploy our flask application to a cloud server. We will use AWS Elastic Beanstalk to deploy our Flask application.

Firstly, we need to create a '.ebextensions' folder in our repository. In that we need to add a python.config file as per below. This will be required by Codepipeline to fetch our application from our GitHub repository and deploy it on Elastic Beanstalk.

.ebextensions > python.config

```
option_settings:
  aws:elasticbeanstalk:container:python:
    WSGIPath: application:application
```

Figure 7.1: python.config code

We will first login to AWS Management Console and open Elastic Beanstalk. Then we will create a sample application with the name 'sentiment-analysis'.

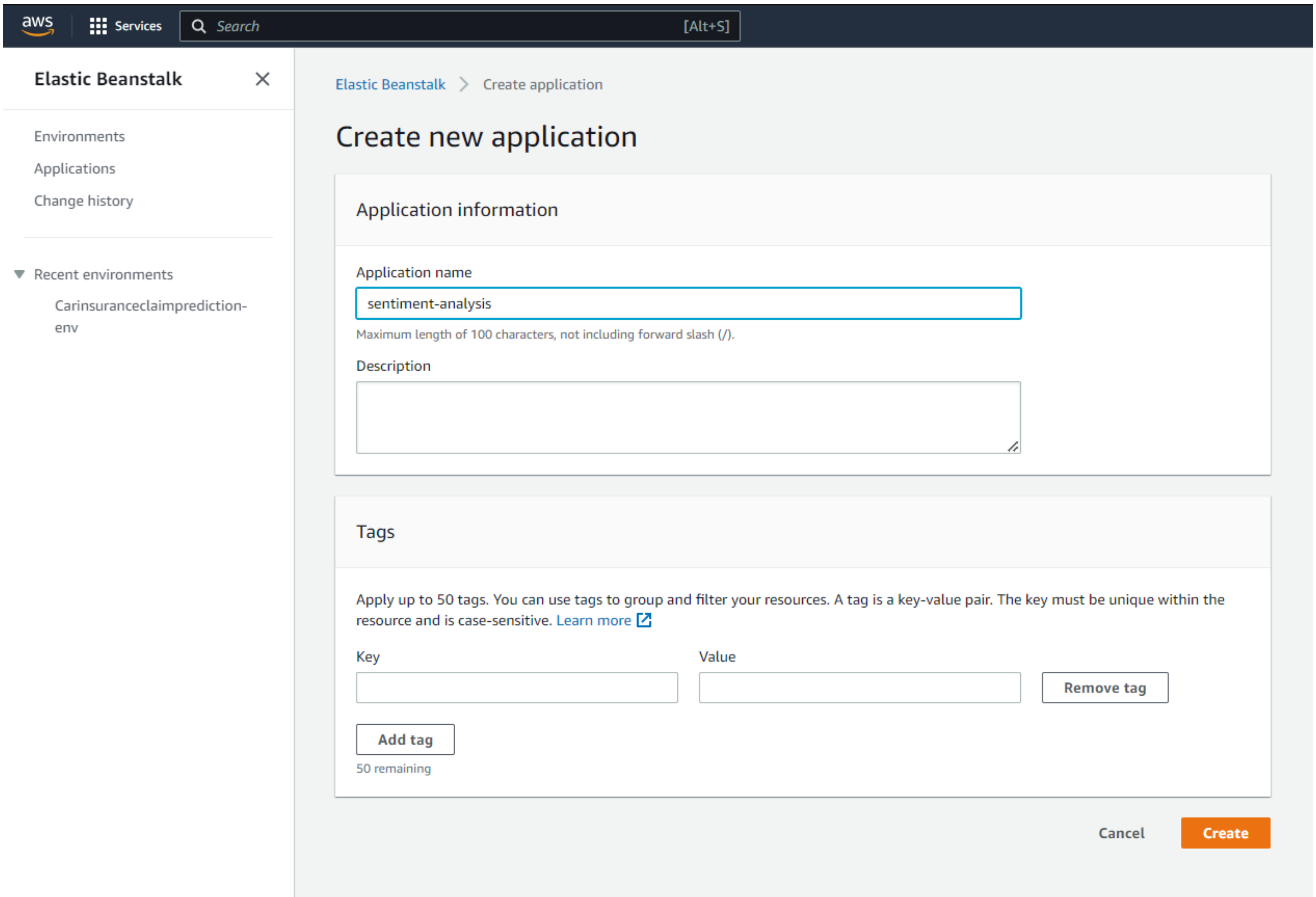

Figure 7.2: Create application window on AWS Elastic Beanstalk

After creating the application, we will have to create a new environment for this application. We will click on 'Create a new environment' button and then select the Web server environment option. We then need to select Platform (which will be Python for our case) and then click Create environment.

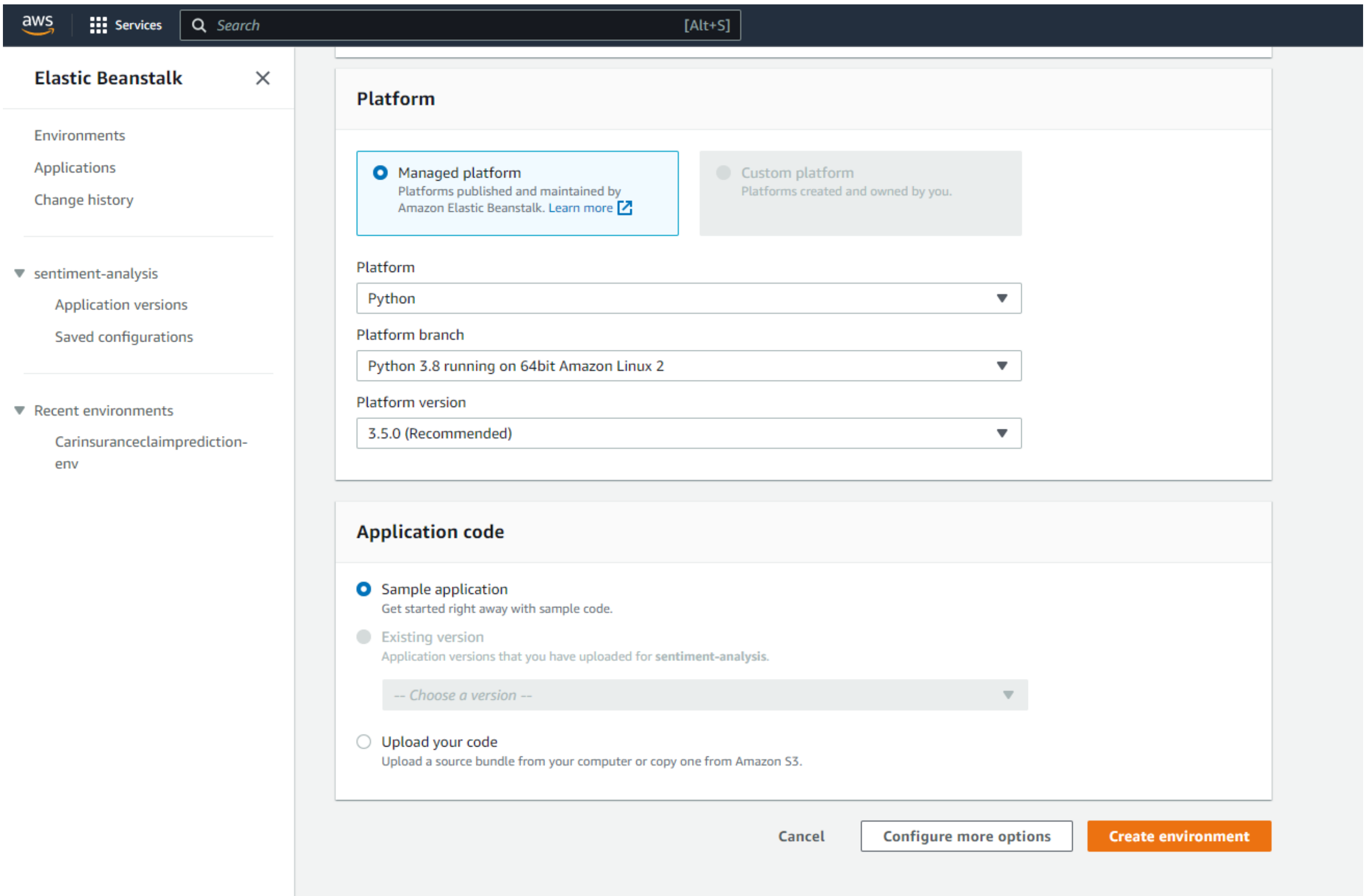

Figure 7.3: Create environment window on AWS Elastic Beanstalk

Creating environment step will take few minutes to complete. Once the environment is created, our application window will look something like this:

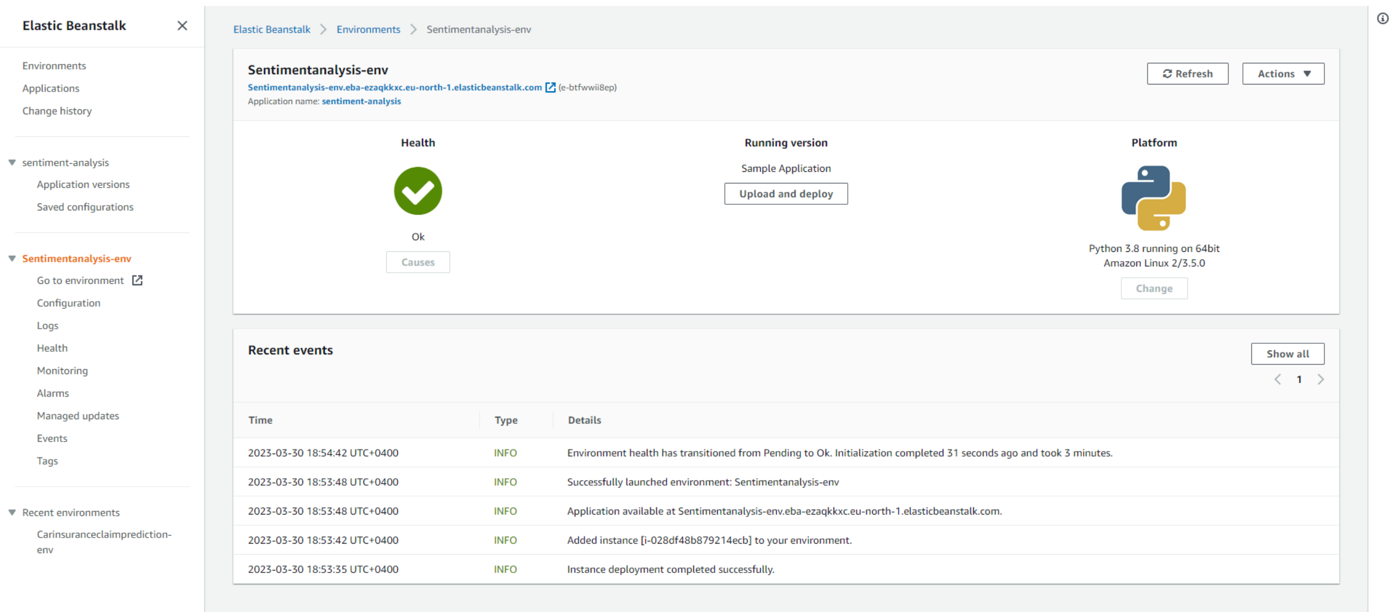

Figure 7.4: New environment window on AWS Elastic Beanstalk

Now, we need to create a pipeline that will link our GitHub repository to this sample application. We

will use Codepipeline provided by AWS for this purpose.
Once we go to Codepipeline using AWS Management Console, we need to click on Create Pipeline. Then we need to give a pipeline name (which we will give 'sentiment-pipeline') and hit Next. We then need to select GitHub (version 2) as the source provider and link our repository as per the image below.

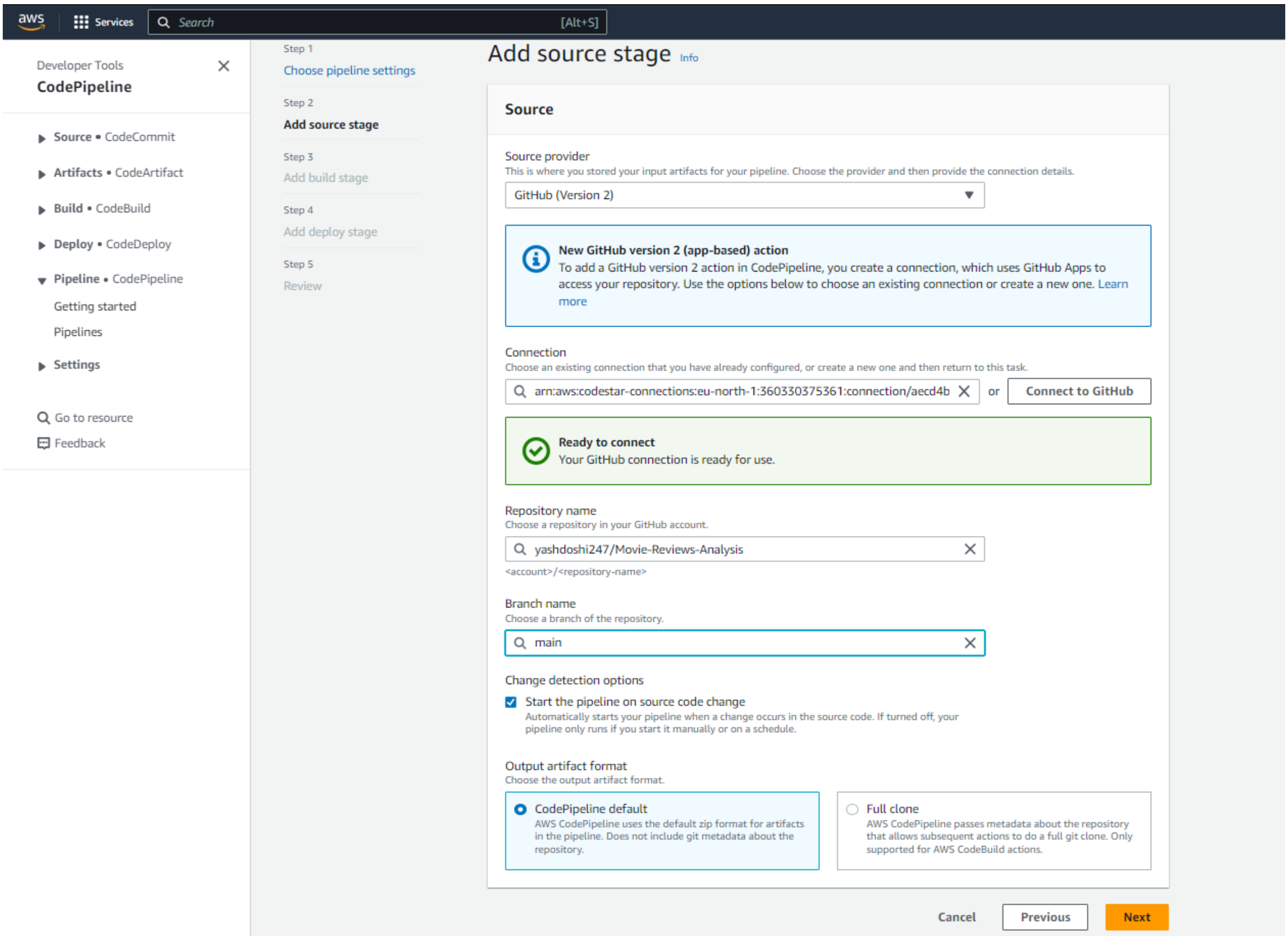

Figure 7.5: Add Source Stage in AWS Codepipeline

After clicking Next, we can skip the 'Add build stage' for our purpose. We then need to add our AWS sample application we created above to the 'Add deploy stage' as per image below.

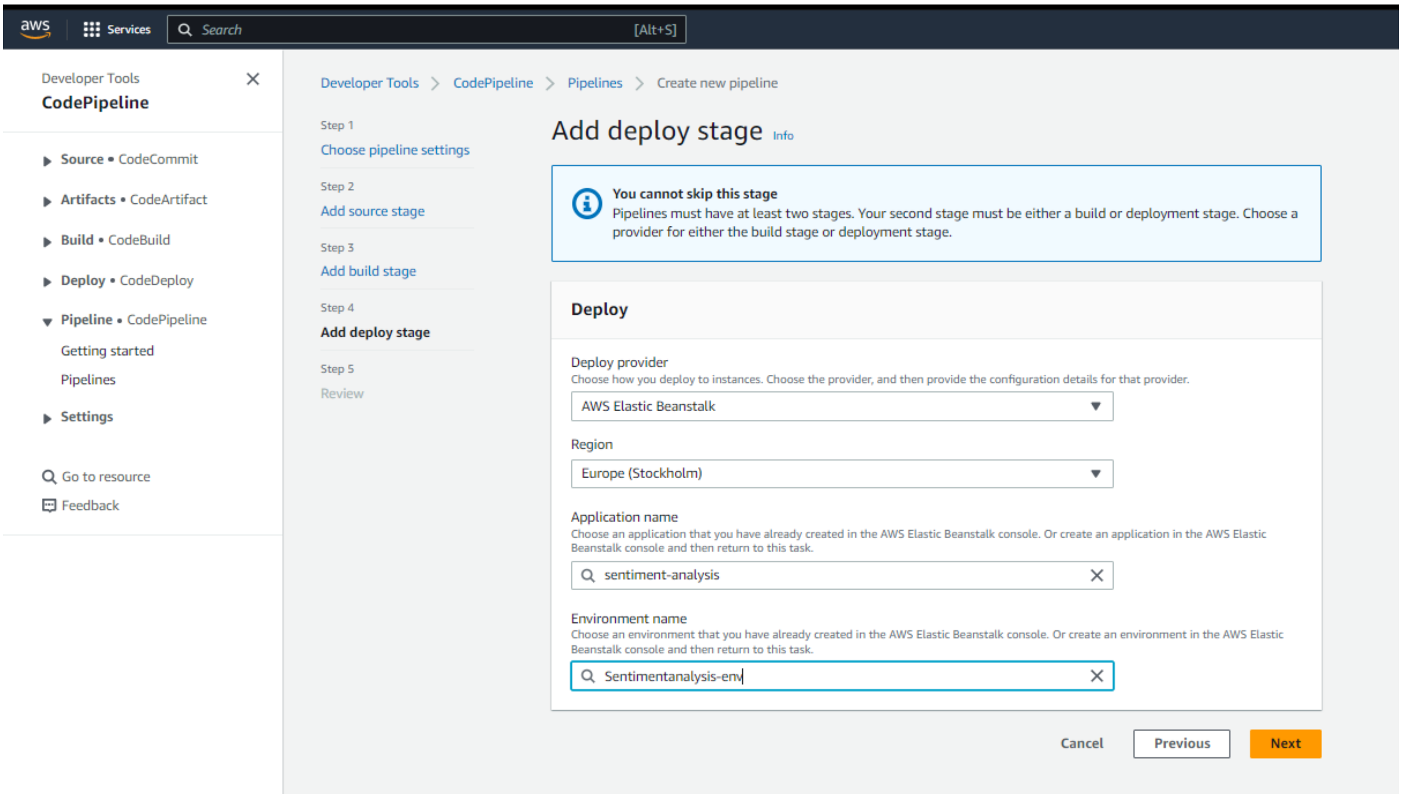

Figure 7.6: Add Deploy Stage in AWS Codepipeline

Then, we can click Next and then click 'Create pipeline'. Once the pipeline gets created, AWS will automatically fetch the latest commit from our GitHub repository and deploy it on AWS Elastic Beanstalk.

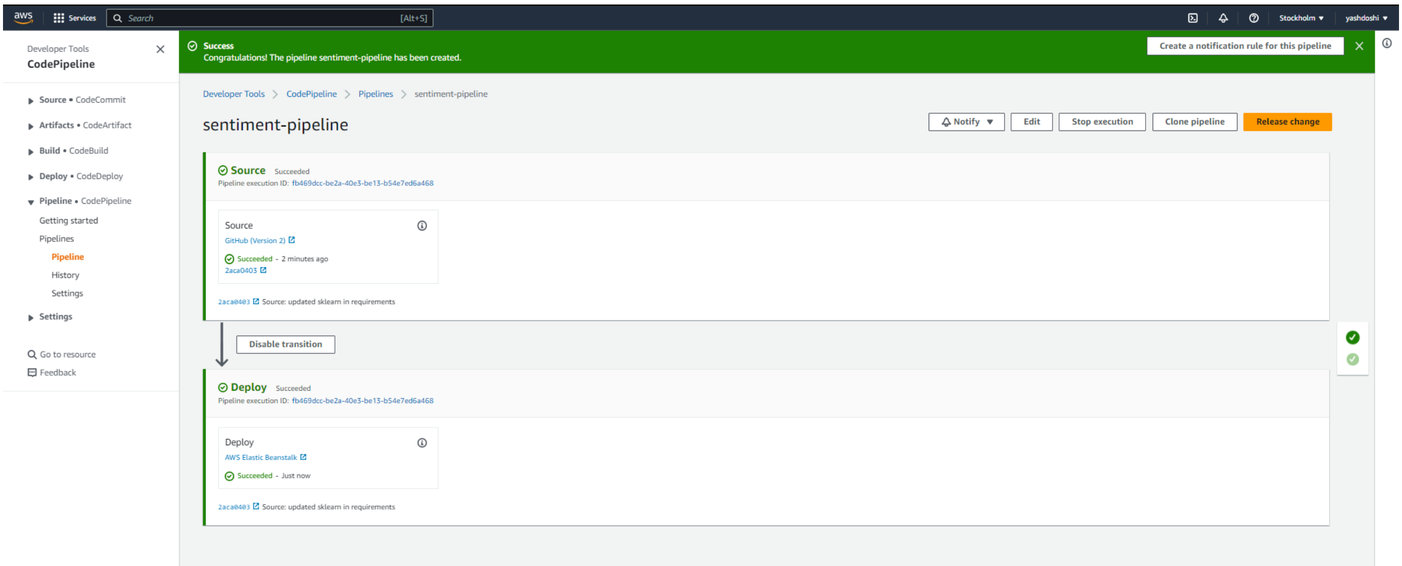

Figure 7.7: Successful creation of AWS Codepipeline

After deployment, this is how the Application pane on Elastic Beanstalk will look,

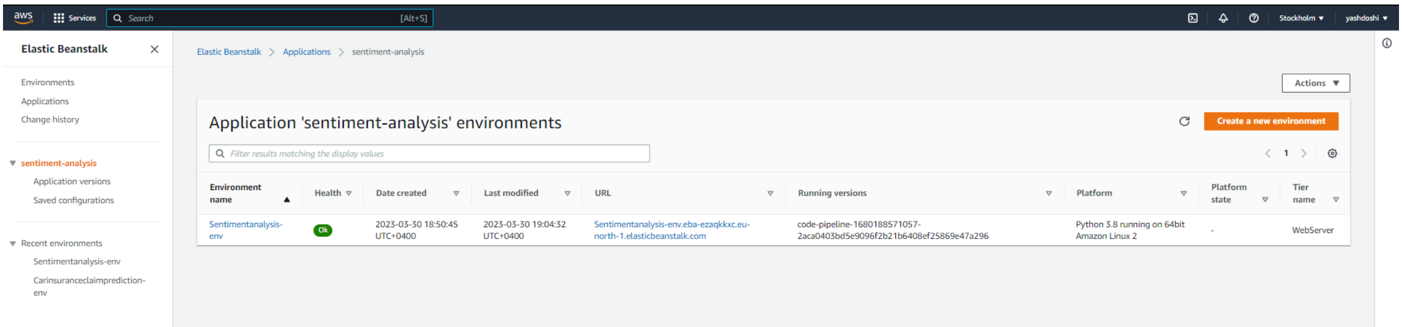

Figure 7.8: Successful deployment of Flask app on AWS Elastic Beanstalk

Health OK signifies that our application is working properly. If our code had any error, it would show 'Degraded' or 'Severe' instead.

Here's the application URL deployed on AWS Elastic Beanstalk:

http://sentimentanalysis-env.eba-ezaqkkxc.eu-north-1.elasticbeanstalk.com/