Deploying a Machine Learning Model as a REST API.

Artwork by Igor Kozak

As a Python developer and data scientist, I have a desire to build web apps to showcase my work. As much as I like to design the front-end, it becomes very overwhelming to take both machine learning and app development. So, I had to find a solution that could easily integrate my machine learning models with other developers who could build a robust web app better than I can.

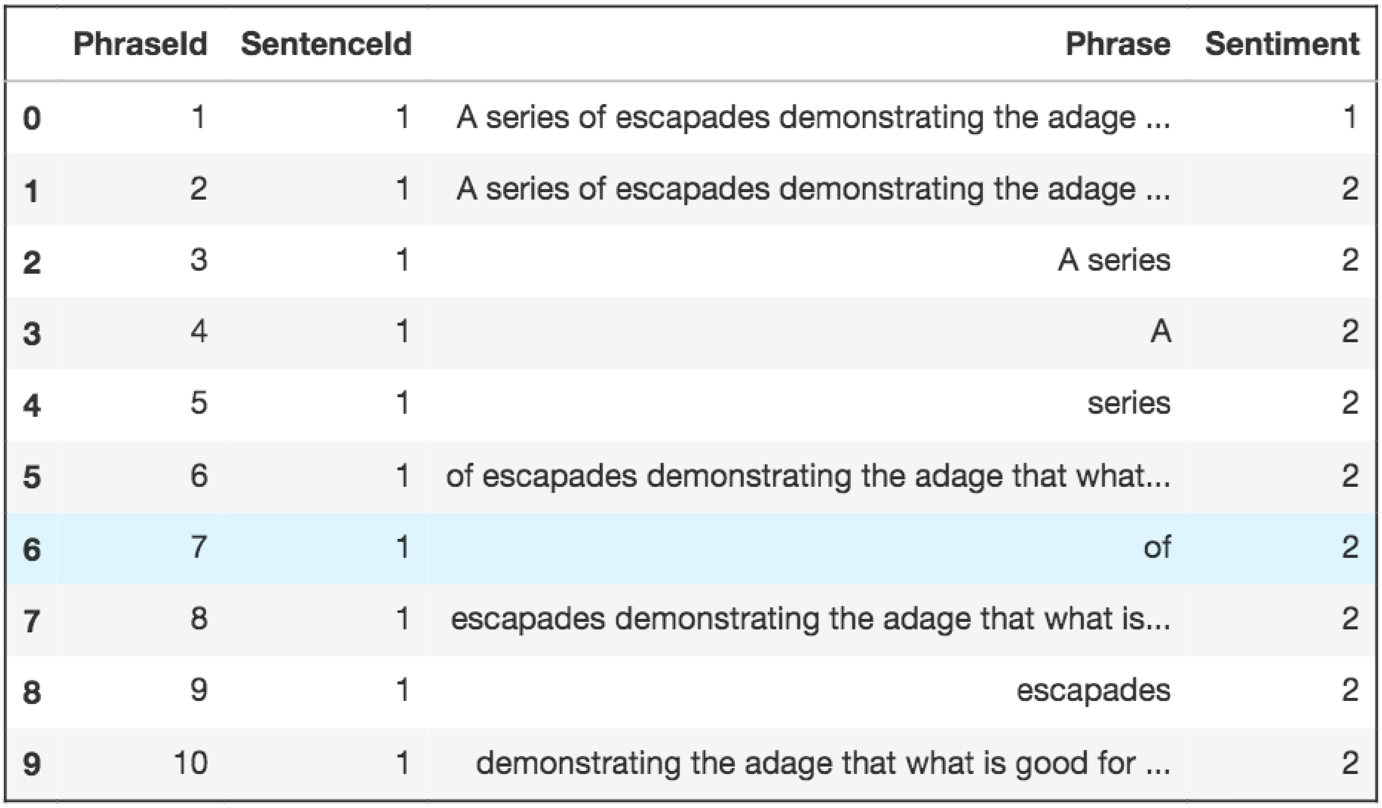
By building a REST API for my model, I could keep my code separate from other developers. There is a clear division of labor here which is nice for defining responsibilities and prevents me from directly blocking teammates who are not involved with the machine learning aspect of the project. Another advantage is that my model can be used by multiple developers working on different platforms, such as web or mobile.

In this article, I will build a simple Scikit-Learn model and deploy it as a REST API using Flask RESTful. This article is intended especially for data scientists who do not have an extensive computer science background.

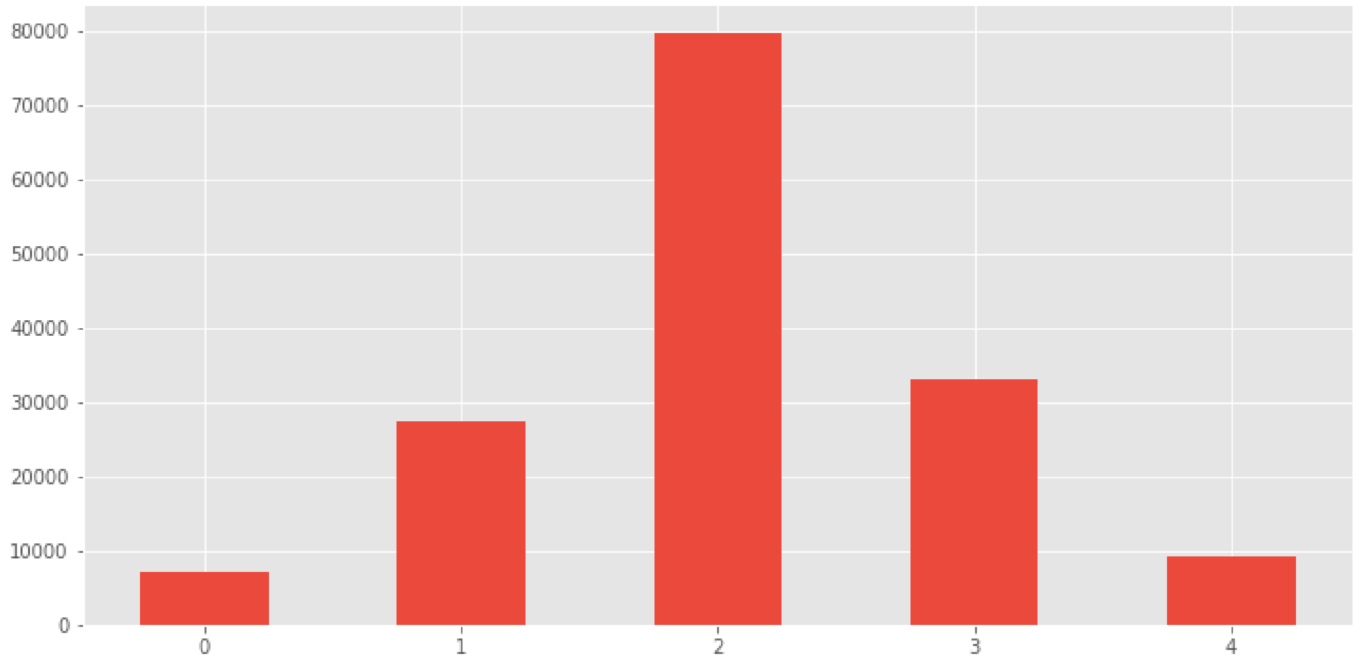
About the Model

For this example, I put together a simple Naives Bayes classifier to predict the sentiment of phrases found in movie reviews.

The data came from the Kaggle competition, Sentiment Analysis on Movie Reviews. The reviews are divided into separate sentences and sentences are further divided into separate phrases. All phrases have a sentiment score so that a model can be trained on which words lend a positive, neutral, or negative sentiment to a sentence.

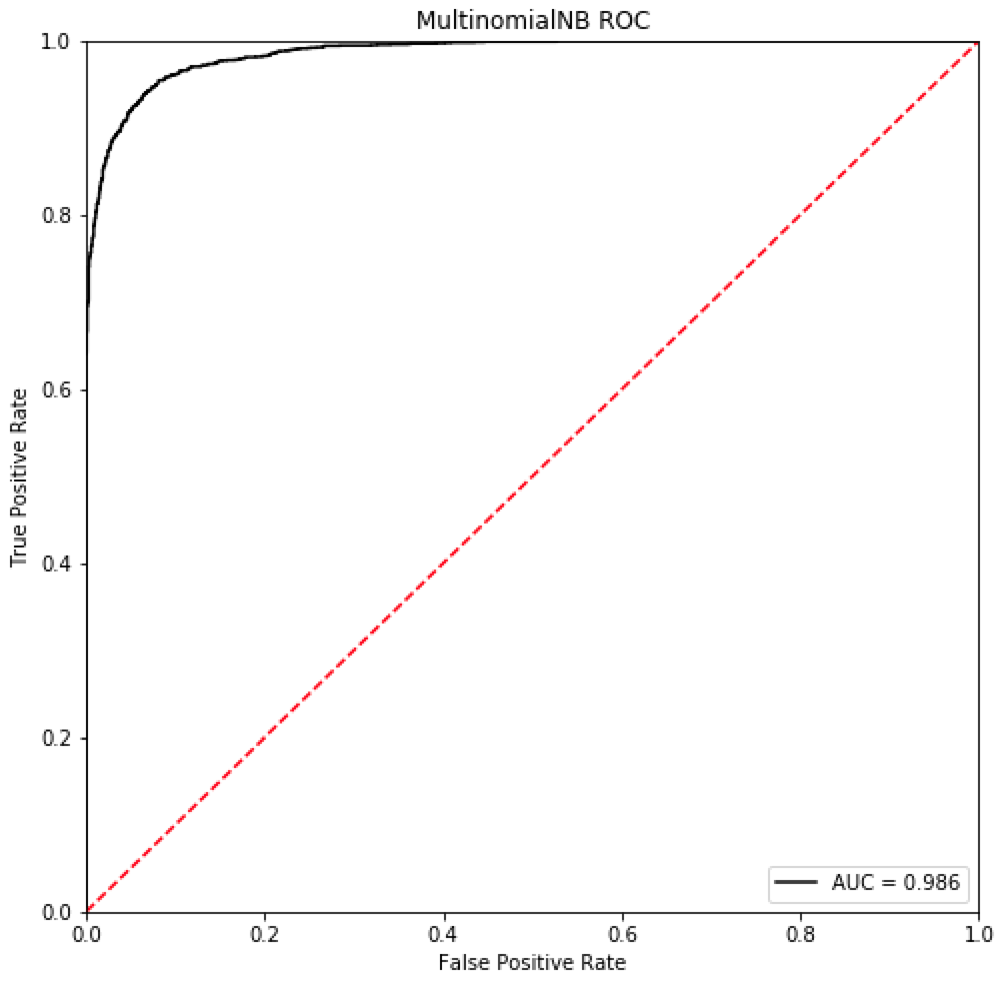


	PhraseId	SentenceId	Phrase	Sentiment
0	1	1	A series of escapades demonstrating the adage ...	1
1	2	1	A series of escapades demonstrating the adage ...	2
2	3	1	A series	2
3	4	1	A	2
4	5	1	series	2
5	6	1	of escapades demonstrating the adage that what...	2
6	7	1	of	2
7	8	1	escapades demonstrating the adage that what is...	2
8	9	1	escapades	2
9	10	1	demonstrating the adage that what is good for ...	2



Distribution of ratings from the Kaggle dataset

The majority of phrases had a neutral rating. At first, I tried to use a multinomial Naive Bayes classifier to predict one out of the 5 possible classes. However, because the majority of the data had a rating of 2, the model did not perform very well. I decided to keep it simple because the main point of this exercise is primarily about deploying as a REST API. So, I limited the data to the extreme classes and trained the model to predict only negative or positive sentiment.



It turned out that the multinomial Naive Bayes model was very effective at predicting positive and negative sentiment. You can find a quick overview of the model training process in this Jupyter Notebook Walkthrough. After training the model in a Jupyter notebook, I transferred my code into Python scripts and created a class object for the

NLP model. You can find the code in my Github repo at this link. You will also need to pickle or save your model so that you can quickly load the trained model into your API script.

Now that we have the model, let's deploy this as a REST API.

REST API Guide

Start a new Python script for your Flask app for the API.

Import Libraries and Load Pickles

The code block below contains a lot of Flask boilerplate and the code to load the classifier and vectorizer pickles.

```
from flask import Flask
from flask_restful import reqparse, abort, Api, Resource
import pickle
import numpy as np
from model import NLPModelapp = Flask(__name__)
api = Api(app)# create new model object
model = NLPModel()# load trained classifier
clf_path = 'lib/models/SentimentClassifier.pkl'
with open(clf_path, 'rb') as f:
    model.clf = pickle.load(f)# load trained vectorizer
vec_path = 'lib/models/TFIDFVectorizer.pkl'
with open(vec_path, 'rb') as f:
    model.vectorizer = pickle.load(f)
```

Create an argument parser

The parser will look through the parameters that a user sends to your API. The parameters will be in a Python dictionary or JSON object. For this example, we will be specifically looking for a key called `query`. The query will be a phrase that a user will want our model to make a prediction on whether the phrase is positive or negative.

```
# argument parsing
parser = reqparse.RequestParser()
parser.add_argument('query')
```

Resource Class Object

Resources are the main building blocks for Flask RESTful APIs. Each class can have methods that correspond to HTTP methods such as: `GET`, `PUT`, `POST`, and `DELETE`. `GET` will be the primary method because our objective is to serve predictions. In the `get` method below, we provide directions on how to handle the user's query and how to package the JSON object that will be returned to the user.

```
class PredictSentiment(Resource):
    def get(self):
        # use parser and find the user's query
        args = parser.parse_args()
        user_query = args['query']        # vectorize the user's
query and make a prediction
        uq_vectorized = model.vectorizer_transform(
            np.array([user_query]))
        prediction = model.predict(uq_vectorized)
        pred_proba = model.predict_proba(uq_vectorized)        #
Output 'Negative' or 'Positive' along with the score
        if prediction == 0:
            pred_text = 'Negative'
        else:
            pred_text = 'Positive'

        # round the predict proba value and set to new variable
        confidence = round(pred_proba[0], 3)        # create JSON
object
        output = {'prediction': pred_text, 'confidence':
confidence}

        return output
```

There is a great tutorial by Flask-RESTful where they build a to-do application and demonstrate how to use the `PUT`, `POST`, and `DELETE` methods.

Endpoints

The following code will set the base url to the sentiment predictor resource. You can imagine that you might have multiple endpoints, each one pointing to a different model that would make different predictions. One example could be an endpoint, `'/ratings'`, which would direct the user to another model that can predict movie ratings given genre, budget, and production members. You would need to create another resource object for this second model. These can just be added right after one another as shown below.

```
api.add_resource(PredictSentiment, '/')

# example of another endpoint
api.add_resource(PredictRatings, '/ratings')
```

Name == Main Block

Not much to say here. Set debug to False if you are deploying this API to production.

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

User Requests

Below are some examples of how users can access your API so that they can get predictions.

With the Requests module in a Jupyter Notebook:

```
url = 'http://127.0.0.1:5000/'
params ={'query': 'that movie was boring'}
response = requests.get(url, params)
response.json()Output: {'confidence': 0.128, 'prediction':
'Negative'}
```

Deployment

Once you have built your model and REST API and finished testing locally, you can deploy your API just as you would any Flask app to the many hosting services on the web. By deploying on the web, users everywhere can make requests to your URL to get predictions. Guides for deployment are included in the Flask docs.

Closing

This was only a very simple example of building a Flask REST API for a sentiment classifier. The same process can be applied to other machine learning or deep learning models once you have trained and saved them.

In addition to deploying models as REST APIs, I am also using REST APIs to manage database queries for data that I have collected by scraping from the web. This lets me collaborate with a full-stack developer without having to manage the code for their React application. If a mobile developer wants to build an app, then they would only have to become familiar with the API endpoints.