Creating a Sentiment Analysis Web App

Using PyTorch and SageMaker

Deep Learning Nanodegree Program | Deployment

Now that we have a basic understanding of how SageMaker works we will try to use it to construct a c
goal will be to have a simple web page which a user can use to enter a movie review. The web page w
model which will predict the sentiment of the entered review.

Instructions

Some template code has already been provided for you, and you will need to implement additional fur
notebook. You will not need to modify the included code beyond what is requested. Sections that begi
you need to complete or implement some portion within them. Instructions will be provided for each s
implementation are marked in the code block with a `# TODO: ...` comment. Please be sure to read t

In addition to implementing code, there will be questions for you to answer which relate to the task an
where you will answer a question is preceded by a '**Question:**' header. Carefully read each question an
header by editing the Markdown cell.

> **Note**: Code and Markdown cells can be executed using the **Shift+Enter** keyboard shortcut. In add
> typically clicking it (double-click for Markdown cells) or by pressing **Enter** while it is highlighted.

General Outline

Recall the general outline for SageMaker projects using a notebook instance.

1. Download or otherwise retrieve the data.
2. Process / Prepare the data.
3. Upload the processed data to S3.
4. Train a chosen model.
5. Test the trained model (typically using a batch transform job).
6. Deploy the trained model.
7. Use the deployed model.

For this project, you will be following the steps in the general outline with some modifications.

First, you will not be testing the model in its own step. You will still be testing the model, however, you
then using the deployed model by sending the test data to it. One of the reasons for doing this is so th
model is working correctly before moving forward.

In addition, you will deploy and use your trained model a second time. In the second iteration you will (
is deployed by including some of your own code. In addition, your newly deployed model will be used i

Step 1: Downloading the data

As in the XGBoost in SageMaker notebook, we will be using the [IMDb dataset]

> Maas, Andrew L., et al. [Learning Word Vectors for Sentiment Analysis]. In *Proceedings of the 49th*
> *Association for Computational Linguistics: Human Language Technologies*. Association for Comp

```
%mkdir ../data
!wget -O ../data/aclImdb_v1.tar.gz http://ai.stanford.edu/~amaas/data/sentiment/aclImc
!tar -zxf ../data/aclImdb_v1.tar.gz -C ../data
```

```
mkdir: cannot create directory '../data': File exists
--2020-05-25 10:32:04--  http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.
Resolving ai.stanford.edu (ai.stanford.edu)... 171.64.68.10
Connecting to ai.stanford.edu (ai.stanford.edu)|171.64.68.10|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 84125825 (80M) [application/x-gzip]
Saving to: '../data/aclImdb_v1.tar.gz'

../data/aclImdb_v1. 100%[===================>]  80.23M  23.5MB/s    in 4.0s

2020-05-25 10:32:08 (19.8 MB/s) - '../data/aclImdb_v1.tar.gz' saved [84125825/841
```

Step 2: Preparing and Processing the data

Also, as in the XGBoost notebook, we will be doing some initial data processing. The first few steps ar
To begin with, we will read in each of the reviews and combine them into a single input structure. Ther
set and a testing set.

```
import os
import glob

def read_imdb_data(data_dir='../data/aclImdb'):
    data = {}
    labels = {}

    for data_type in ['train', 'test']:
        data[data_type] = {}
        labels[data_type] = {}

        for sentiment in ['pos', 'neg']:
            data[data_type][sentiment] = []
```

```
            labels[data_type][sentiment] = []

            path = os.path.join(data_dir, data_type, sentiment, '*.txt')
            files = glob.glob(path)

            for f in files:
                with open(f) as review:
                    data[data_type][sentiment].append(review.read())
                    # Here we represent a positive review by '1' and a negative reviev
                    labels[data_type][sentiment].append(1 if sentiment == 'pos' else (

            assert len(data[data_type][sentiment]) == len(labels[data_type][sentiment]
                    "{}/{} data size does not match labels size".format(data_type, ser

    return data, labels


data, labels = read_imdb_data()
print("IMDB reviews: train = {} pos / {} neg, test = {} pos / {} neg".format(
            len(data['train']['pos']), len(data['train']['neg']),
            len(data['test']['pos']), len(data['test']['neg'])))
```

```
IMDB reviews: train = 12500 pos / 12500 neg, test = 12500 pos / 12500 neg
```

Now that we've read the raw training and testing data from the downloaded dataset, we will combine t
shuffle the resulting records.

```
from sklearn.utils import shuffle

def prepare_imdb_data(data, labels):
    """Prepare training and test sets from IMDb movie reviews."""

    #Combine positive and negative reviews and labels
    data_train = data['train']['pos'] + data['train']['neg']
    data_test = data['test']['pos'] + data['test']['neg']
    labels_train = labels['train']['pos'] + labels['train']['neg']
    labels_test = labels['test']['pos'] + labels['test']['neg']

    #Shuffle reviews and corresponding labels within training and test sets
    data_train, labels_train = shuffle(data_train, labels_train)
    data_test, labels_test = shuffle(data_test, labels_test)

    # Return a unified training data, test data, training labels, test labets
    return data_train, data_test, labels_train, labels_test


train_X, test_X, train_y, test_y = prepare_imdb_data(data, labels)
print("IMDb reviews (combined): train = {}, test = {}".format(len(train_X), len(test_X
```

```
IMDb reviews (combined): train = 25000, test = 25000
```

Now that we have our training and testing sets unified and prepared, we should do a quick check and be trained on. This is generally a good idea as it allows you to see how each of the further processing ensures that the data has been loaded correctly.

```
print(train_X[100])
print(train_y[100])
```

```
All I could think of while watching this movie was B-grade slop. Many have spoken
0
```

The first step in processing the reviews is to make sure that any html tags that appear should be remo input, that way words such as *entertained* and *entertaining* are considered the same with regard to ser

```
import nltk
from nltk.corpus import stopwords
from nltk.stem.porter import *

import re
from bs4 import BeautifulSoup

def review_to_words(review):
    nltk.download("stopwords", quiet=True)
    stemmer = PorterStemmer()

    text = BeautifulSoup(review, "html.parser").get_text() # Remove HTML tags
    text = re.sub(r"[^a-zA-Z0-9]", " ", text.lower()) # Convert to lower case
    words = text.split() # Split string into words
    words = [w for w in words if w not in stopwords.words("english")] # Remove stopwo
    words = [PorterStemmer().stem(w) for w in words] # stem

    return words
```

The `review_to_words` method defined above uses `BeautifulSoup` to remove any html tags that ap tokenize the reviews. As a check to ensure we know how everything is working, try applying `review_t` training set.

```
# TODO: Apply review_to_words to a review (train_X[100] or any other review)
review_to_words(train_X[220])
```

```
['get',
 'amaz',
 'bad',
 'film',
 'world',
 'anybodi',
 'could',
 'rais',
 'money',
 'make',
 'kind',
 'crap',
 'absolut',
 'talent',
 'includ',
 'film',
 'crappi'
```

Question: Above we mentioned that `review_to_words` method removes html formatting and allows for example, converting *entertained* and *entertaining* into *entertain* so that they are treated as though t anything, does this method do to the input?

```
 'amaz']
```

Answer:

The method below applies the `review_to_words` method to each of the reviews in the training and te results. This is because performing this processing step can take a long time. This way if you are unal current session, you can come back without needing to process the data a second time.

```
import pickle

cache_dir = os.path.join("../cache", "sentiment_analysis")  # where to store cache fil
os.makedirs(cache_dir, exist_ok=True)  # ensure cache directory exists

def preprocess_data(data_train, data_test, labels_train, labels_test,
                    cache_dir=cache_dir, cache_file="preprocessed_data.pkl"):
    """Convert each review to words; read from cache if available."""

    # If cache_file is not None, try to read from it first
    cache_data = None
    if cache_file is not None:
        try:
            with open(os.path.join(cache_dir, cache_file), "rb") as f:
                cache_data = pickle.load(f)
            print("Read preprocessed data from cache file:", cache_file)
        except:
            pass  # unable to read from cache, but that's okay

    # If cache is missing, then do the heavy lifting
    if cache_data is None:
        # Preprocess training and test data to obtain words for each review
        #words_train = list(map(review_to_words, data_train))
```

```
        #words_train = list(map(review_to_words, data_train))
        #words_test = list(map(review_to_words, data_test))
        words_train = [review_to_words(review) for review in data_train]
        words_test = [review_to_words(review) for review in data_test]

        # Write to cache file for future runs
        if cache_file is not None:
            cache_data = dict(words_train=words_train, words_test=words_test,
                              labels_train=labels_train, labels_test=labels_test)
            with open(os.path.join(cache_dir, cache_file), "wb") as f:
                pickle.dump(cache_data, f)
            print("Wrote preprocessed data to cache file:", cache_file)
    else:
        # Unpack data loaded from cache file
        words_train, words_test, labels_train, labels_test = (cache_data['words_train'
                cache_data['words_test'], cache_data['labels_train'], cache_data['labe

    return words_train, words_test, labels_train, labels_test


# Preprocess data
train_X, test_X, train_y, test_y = preprocess_data(train_X, test_X, train_y, test_y)
```

```
Read preprocessed data from cache file: preprocessed_data.pkl
```

Transform the data

In the XGBoost notebook we transformed the data from its word representation to a bag-of-words fea going to construct in this notebook we will construct a feature representation which is very similar. To integer. Of course, some of the words that appear in the reviews occur very infrequently and so likely c purposes of sentiment analysis. The way we will deal with this problem is that we will fix the size of ou include the words that appear most frequently. We will then combine all of the infrequent words into a label it as `1`.

Since we will be using a recurrent neural network, it will be convenient if the length of each review is th our reviews and then pad short reviews with the category 'no word' (which we will label `0`) and trunca

(TODO) Create a word dictionary

To begin with, we need to construct a way to map words that appear in the reviews to integers. Here w the 'no word' and 'infrequent' categories) to be `5000` but you may wish to change this to see how it af

> **TODO:** Complete the implementation for the `build_dict()` method below. Note that even thoug we only want to construct a mapping for the most frequently appearing `4998` words. This is beca special labels `0` for 'no word' and `1` for 'infrequent word'.

↳ *7 cells hidden*

Save `word_dict`

Later on when we construct an endpoint which processes a submitted review we will need to make us
created. As such, we will save it to a file now for future use.

↳ 2 cells hidden

Transform the reviews

Now that we have our word dictionary which allows us to transform the words appearing in the review
and convert our reviews to their integer sequence representation, making sure to pad or truncate to a

↳ 6 cells hidden

Step 3: Upload the data to S3

As in the XGBoost notebook, we will need to upload the training dataset to S3 in order for our training
locally and we will upload to S3 later on.

Save the processed training dataset locally

It is important to note the format of the data that we are saving as we will need to know it when we wr
row of the dataset has the form `label`, `length`, `review[500]` where `review[500]` is a sequence o
the review.

```
import pandas as pd

pd.concat([pd.DataFrame(train_y), pd.DataFrame(train_X_len), pd.DataFrame(train_X)], a
        .to_csv(os.path.join(data_dir, 'train.csv'), header=False, index=False)
```

Uploading the training data

Next, we need to upload the training data to the SageMaker default S3 bucket so that we can provide a

↳ 3 cells hidden

Step 4: Build and Train the PyTorch Model

In the XGBoost notebook we discussed what a model is in the SageMaker framework. In particular, a r

- Model Artifacts,
- Training Code, and
- Inference Code,

each of which interact with one another. In the XGBoost example we used training and inference code will still be using containers provided by Amazon with the added benefit of being able to include our o

We will start by implementing our own neural network in PyTorch along with a training script. For the p the necessary model object in the `model.py` file, inside of the `train` folder. You can see the providec below.

```
!pygmentize train/model.py
```

```python
import torch.nn as nn

class LSTMClassifier(nn.Module):
    """
    This is the simple RNN model we will be using to perform Sentiment Analysis.
    """

    def __init__(self, embedding_dim, hidden_dim, vocab_size):
        """
        Initialize the model by settingg up the various layers.
        """
        super(LSTMClassifier, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)
        self.dense = nn.Linear(in_features=hidden_dim, out_features=1)
        self.sig = nn.Sigmoid()

        self.word_dict = None

    def forward(self, x):
        """
        Perform a forward pass of our model on some input.
        """
        x = x.t()
        lengths = x[0,:]
        reviews = x[1:,:]
        embeds = self.embedding(reviews)
        lstm_out, _ = self.lstm(embeds)
        out = self.dense(lstm_out)
        out = out[lengths - 1, range(len(lengths))]
        return self.sig(out.squeeze())
```

The important takeaway from the implementation provided is that there are three parameters that we performance of our model. These are the embedding dimension, the hidden dimension and the size o make these parameters configurable in the training script so that if we wish to modify them we do not see how to do this later on. To start we will write some of the training code in the notebook so that we arise.

First we will load a small portion of the training data set to use as a sample. It would be very time con completely in the notebook as we do not have access to a gpu and the compute instance that we are

However, we can work on a small bit of the data to get a feel for how our training script is behaving.

```
import torch
import torch.utils.data

# Read in only the first 250 rows
train_sample = pd.read_csv(os.path.join(data_dir, 'train.csv'), header=None, names=Nor

# Turn the input pandas dataframe into tensors
train_sample_y = torch.from_numpy(train_sample[[0]].values).float().squeeze()
train_sample_X = torch.from_numpy(train_sample.drop([0], axis=1).values).long()

# Build the dataset
train_sample_ds = torch.utils.data.TensorDataset(train_sample_X, train_sample_y)
# Build the dataloader
train_sample_dl = torch.utils.data.DataLoader(train_sample_ds, batch_size=50)
```

(TODO) Writing the training method

Next we need to write the training code itself. This should be very similar to training methods that you models. We will leave any difficult aspects such as model saving / loading and parameter loading unti

↳ *4 cells hidden*

(TODO) Training the model

When a PyTorch model is constructed in SageMaker, an entry point must be specified. This is the Pyth model is trained. Inside of the `train` directory is a file called `train.py` which has been provided anc code to train our model. The only thing that is missing is the implementation of the `train()` method

TODO: Copy the `train()` method written above and paste it into the `train/train.py` file where req

The way that SageMaker passes hyperparameters to the training script is by way of arguments. These in the training script. To see how this is done take a look at the provided `train/train.py` file.

↳ *2 cells hidden*

Step 5: Testing the model

As mentioned at the top of this notebook, we will be testing this model by first deploying it and then s endpoint. We will do this so that we can make sure that the deployed model is working correctly.

Step 6: Deploy the model for testing

Now that we have trained our model, we would like to test it to see how it performs. Currently our moc `review_length, review[500]` where `review[500]` is a sequence of `500` integers which describe using `word_dict`. Fortunately for us, SageMaker provides built-in inference code for models with sim

There is one thing that we need to provide, however, and that is a function which loads the saved mod `model_fn()` and takes as its only parameter a path to the directory where the model artifacts are sto the python file which we specified as the entry point. In our case the model loading function has been made.

NOTE: When the built-in inference code is run it must import the `model_fn()` method from the `trair` wrapped in a main guard (ie, `if __name__ == '__main__':`)

Since we don't need to change anything in the code that was uploaded during training, we can simply

NOTE: When deploying a model you are asking SageMaker to launch an compute instance that will wa compute instance will continue to run until *you* shut it down. This is important to know since the cost long it has been running for.

In other words **If you are no longer using a deployed endpoint, shut it down!**

TODO: Deploy the trained model.

```
# TODO: Deploy the trained model
predictor = estimator.deploy(initial_instance_count=1, instance_type='ml.m4.xlarge')
```

```
-------------!
```

Step 7 - Use the model for testing

Once deployed, we can read in the test data and send it off to our deployed model to get some results determine how accurate our model is.

```
test_X = pd.concat([pd.DataFrame(test_X_len), pd.DataFrame(test_X)], axis=1)


# We split the data into chunks and send each chunk seperately, accumulating the resul

def predict(data, rows=512):
    split_array = np.array_split(data, int(data.shape[0] / float(rows) + 1))
    predictions = np.array([])
    for array in split_array:
        predictions = np.append(predictions, predictor.predict(array))

    return predictions


predictions = predict(test_X.values)
predictions = [round(num) for num in predictions]


from sklearn.metrics import accuracy_score
accuracy_score(test_y, predictions)
```

```
0.8566
```

Question: How does this model compare to the XGBoost model you created earlier? Why might these dataset? Which do *you* think is better for sentiment analysis?

Answer: it's higher than the XGBoost model so I would choose this one

(TODO) More testing

We now have a trained model which has been deployed and which we can send processed reviews to sentiment. However, ultimately we would like to be able to send our model an unprocessed review. Th itself as a string. For example, suppose we wish to send the following review to our model.

↳ *6 cells hidden*

Delete the endpoint

Of course, just like in the XGBoost notebook, once we've deployed an endpoint it continues to run unti using our endpoint for now, we can delete it.

↳ *1 cell hidden*

Step 6 (again) - Deploy the model for the web app

Now that we know that our model is working, it's time to create some custom inference code so that w not been processed and have it determine the sentiment of the review.

As we saw above, by default the estimator which we created, when deployed, will use the entry script creating the model. However, since we now wish to accept a string as input and our model expects a custom inference code.

We will store the code that we write in the `serve` directory. Provided in this directory is the `model.py` a `utils.py` file which contains the `review_to_words` and `convert_and_pad` pre-processing functi processing, and `predict.py`, the file which will contain our custom inference code. Note also that `re` tell SageMaker what Python libraries are required by our custom inference code.

When deploying a PyTorch model in SageMaker, you are expected to provide four functions which the

- `model_fn`: This function is the same function that we used in the training script and it tells Sage
- `input_fn`: This function receives the raw serialized input that has been sent to the model's end make the input available for the inference code.
- `output_fn`: This function takes the output of the inference code and its job is to serialize this o model's endpoint.

- `predict_fn`: The heart of the inference script, this is where the actual prediction is done and is complete.

For the simple website that we are constructing during this project, the `input_fn` and `output_fn` m only require being able to accept a string as input and we expect to return a single value as output. Yo complex application the input or output may be image data or some other binary data which would re

(TODO) Writing inference code

Before writing our custom inference code, we will begin by taking a look at the code which has been p

```
!pygmentize serve/predict.py
```

```
import argparse
import json
import os
import pickle
import sys
import sagemaker_containers
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data

from model import LSTMClassifier

from utils import review_to_words, convert_and_pad

def model_fn(model_dir):
    """Load the PyTorch model from the `model_dir` directory."""
    print("Loading model.")

    # First, load the parameters used to create the model.
    model_info = {}
    model_info_path = os.path.join(model_dir, 'model_info.pth')
    with open(model_info_path, 'rb') as f:
        model_info = torch.load(f)

    print("model_info: {}".format(model_info))

    # Determine the device and construct the model.
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = LSTMClassifier(model_info['embedding_dim'], model_info['hidden_dim'],

    # Load the store model parameters.
    model_path = os.path.join(model_dir, 'model.pth')
    with open(model_path, 'rb') as f:
        model.load_state_dict(torch.load(f))

    # Load the saved word_dict.
    word_dict_path = os.path.join(model_dir, 'word_dict.pkl')
    with open(word_dict_path, 'rb') as f:
        model.word_dict = pickle.load(f)

    model.to(device).eval()

    print("Done loading model.")
    return model

def input_fn(serialized_input_data, content_type):
    print('Deserializing the input data.')
    if content_type == 'text/plain':
        data = serialized_input_data.decode('utf-8')
        return data
    raise Exception('Requested unsupported ContentType in content_type: ' + conte
```

As mentioned earlier, the `model_fn` method is the same as the one provided in the training code and are very simple and your task will be to complete the `predict_fn` method. Make sure that you save t `serve` directory.

Deploying the model

Now that the custom inference code has been written, we will create and deploy our model. To begin \ PyTorchModel object which points to the model artifacts created during training and also points to th Then we can call the deploy method to launch the deployment container.

NOTE: The default behaviour for a deployed PyTorch model is to assume that any input passed to the want to send a string so we need to construct a simple wrapper around the `RealTimePredictor` clas more complicated situation you may want to provide a serialization object, for example if you wanted

↳ *1 cell hidden*

Testing the model

Now that we have deployed our model with the custom inference code, we should test to see if everyt by loading the first `250` positive and negative reviews and send them to the endpoint, then collect the some of the data is that the amount of time it takes for our model to process the input and then perfo the entire data set would be prohibitive.

↳ *6 cells hidden*

Step 7 (again): Use the model for the web app

> **TODO:** This entire section and the next contain tasks for you to complete, mostly using the AWS

So far we have been accessing our model endpoint by constructing a predictor object which uses the object to perform inference. What if we wanted to create a web app which accessed our model? The \ not possible since in order to access a SageMaker endpoint the app would first have to authenticate \ access to SageMaker endpoints. However, there is an easier way! We just need to use some additiona

The diagram above gives an overview of how the various services will work together. On the far right i which is deployed using SageMaker. On the far left is our web app that collects a user's movie review, negative sentiment in return.

In the middle is where some of the magic happens. We will construct a Lambda function, which you c function that can be executed whenever a specified event occurs. We will give this function permissio SageMaker endpoint.

Lastly, the method we will use to execute the Lambda function is a new endpoint that we will create us url that listens for data to be sent to it. Once it gets some data it will pass that data on to the Lambda Lambda function returns. Essentially it will act as an interface that lets our web app communicate wit

Setting up a Lambda function

The first thing we are going to do is set up a Lambda function. This Lambda function will be executed it. When it is executed it will receive the data, perform any sort of processing that is required, send the endpoint we've created and then return the result.

Part A: Create an IAM Role for the Lambda function

Since we want the Lambda function to call a SageMaker endpoint, we need to make sure that it has pe construct a role that we can later give the Lambda function.

Using the AWS Console, navigate to the **IAM** page and click on **Roles**. Then, click on **Create role**. Make trusted entity selected and choose **Lambda** as the service that will use this role, then click **Next: Perm**

In the search box type `sagemaker` and select the check box next to the **AmazonSageMakerFullAcces**

Lastly, give this role a name. Make sure you use a name that you will remember later on, for example **Create role**.

Part B: Create a Lambda function

Now it is time to actually create the Lambda function.

Using the AWS Console, navigate to the AWS Lambda page and click on **Create a function**. When you **Author from scratch** is selected. Now, name your Lambda function, using a name that you will remem `sentiment_analysis_func`. Make sure that the **Python 3.6** runtime is selected and then choose the Then, click on **Create Function**.

On the next page you will see some information about the Lambda function you've just created. If you which you can write the code that will be executed when your Lambda function is triggered. In our exa

```
# We need to use the low-level library to interact with SageMaker since the SageMaker API
# is not available natively through Lambda.
import boto3

def lambda_handler(event, context):

    # The SageMaker runtime is what allows us to invoke the endpoint that we've created.
    runtime = boto3.Session().client('sagemaker-runtime')

    # Now we use the SageMaker runtime to invoke our endpoint, sending the review we were give
    response = runtime.invoke_endpoint(EndpointName = '**ENDPOINT NAME HERE**',    # The name
                                       ContentType = 'text/plain',                 # The data
```

```
                                  Body = event['body'])                    # The actua

    # The response is an HTTP response whose body contains the result of our inference
    result = response['Body'].read().decode('utf-8')

    return {
        'statusCode' : 200,
        'headers' : { 'Content-Type' : 'text/plain', 'Access-Control-Allow-Origin' : '*' },
        'body' : result
    }
```

Once you have copy and pasted the code above into the Lambda code editor, replace the `**ENDPOINT` the endpoint that we deployed earlier. You can determine the name of the endpoint using the code cel

```
predictor.endpoint
```

```
'sagemaker-pytorch-2020-05-25-10-56-56-377'
```

Once you have added the endpoint name to the Lambda function, click on **Save**. Your Lambda functio create a way for our web app to execute the Lambda function.

Setting up API Gateway

Now that our Lambda function is set up, it is time to create a new API using API Gateway that will trig created.

Using AWS Console, navigate to **Amazon API Gateway** and then click on **Get started**.

On the next page, make sure that **New API** is selected and give the new api a name, for example, `sent` **Create API**.

Now we have created an API, however it doesn't currently do anything. What we want it to do is to trig earlier.

Select the **Actions** dropdown menu and click **Create Method**. A new blank method will be created, sel then click on the check mark beside it.

For the integration point, make sure that **Lambda Function** is selected and click on the **Use Lambda P** that the data that is sent to the API is then sent directly to the Lambda function with no processing. It a proper response object as it will also not be processed by API Gateway.

Type the name of the Lambda function you created earlier into the **Lambda Function** text entry box an pop-up box that then appears, giving permission to API Gateway to invoke the Lambda function you c

The last step in creating the API Gateway is to select the **Actions** dropdown and click on **Deploy API**. ` stage and name it anything you like, for example `prod`.

You have now successfully set up a public API to access your SageMaker model. Make sure to copy c your newly created public API as this will be needed in the next step. This URL can be found at the top the text **Invoke URL**.

Step 4: Deploying our web app

Now that we have a publicly available API, we can start using it in a web app. For our purposes, we ha can make use of the public api you created earlier.

In the `website` folder there should be a file called `index.html`. Download the file to your computer a choice. There should be a line which contains ****REPLACE WITH PUBLIC API URL****. Replace this strin last step and then save the file.

Now, if you open `index.html` on your local computer, your browser will behave as a local web server interact with your SageMaker model.

If you'd like to go further, you can host this html file anywhere you'd like, for example using github or h you have done this you can share the link with anyone you'd like and have them play with it too!

> **Important Note** In order for the web app to communicate with the SageMaker endpoint, the endp and running. This means that you are paying for it. Make sure that the endpoint is running when y that you shut it down when you don't need it, otherwise you will end up with a surprisingly large A

TODO: Make sure that you include the edited `index.html` file in your project submission.

Now that your web app is working, trying playing around with it and see how well it works.

Question: Give an example of a review that you entered into your web app. What was the predicted se

Answer: "The cinematography was atrocious", the predicted sentiment was negative

Delete the endpoint

Remember to always shut down your endpoint if you are no longer using it. You are charged for the le so if you forget and leave it on you could end up with an unexpectedly large bill.

↳ *2 cells hidden*