**Introduction:**

Sentiment analysis, also refers as opinion mining is a sub machine learning task where the general sentiment of a given document is determined. Using Machine Learning and Natural language processing (NLP) subjective information of a document is extracted and classified according to its polarity (Positive or Negative). It is a useful analysis as it provides an overall opinion of the viewer about a movie. Sentiment analysis is far from to be solved as the language is very complex. This complexity makes it interesting.

In this project, I choose to try to classify the reviews from IMDB Dataset into 'positive' or 'negative' sentiment by building a model based on neural networks. IMDB is an online database of information related to films, television etc., Here viewers share their opinion about any movie they watched. It is perfect source of data to determine the current overall opinion about any movie.

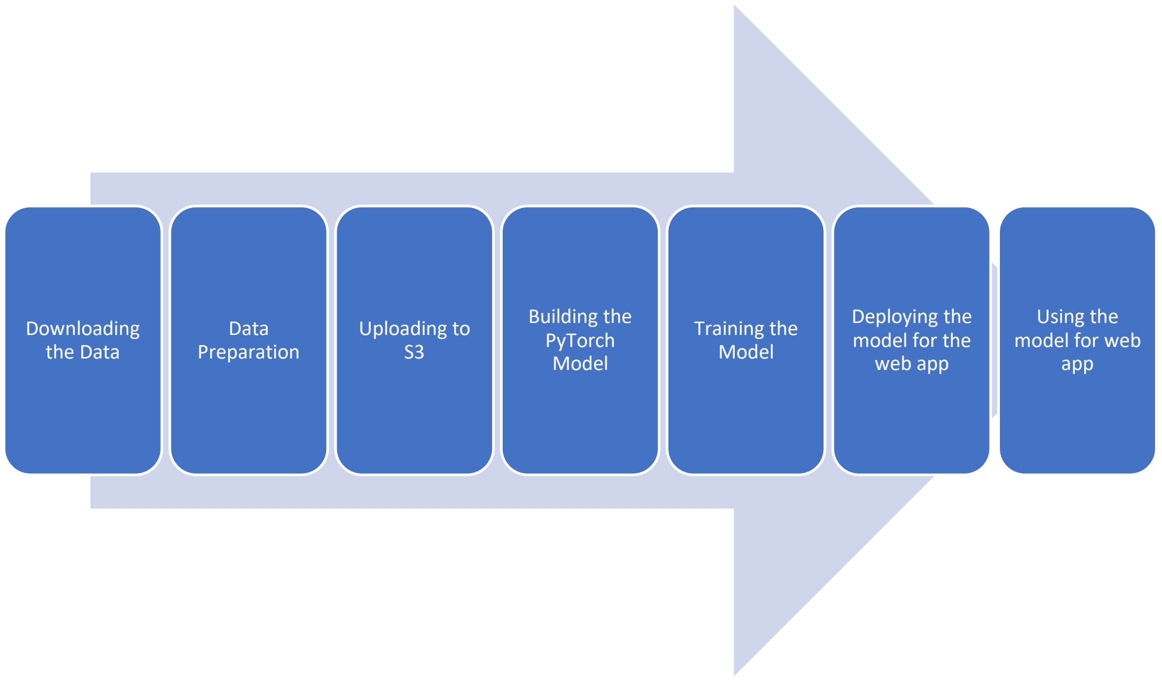
**Data Sourcing:**

<u>IMDB Dataset:</u>

This dataset contains reviews of movies from **IMBD**. This dataset contains variables such as review about a movie and label indicating whether the review is positive or negative. This is a dataset for binary sentiment classification. It contains 25,000 highly polar movie reviews for training and 25,000 for testing. The dataset is taken from ai.standford.edu.

**Methodology:**

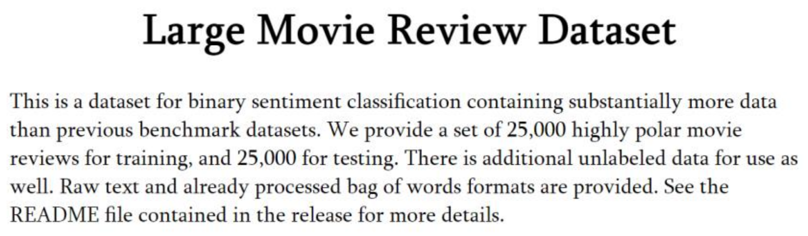
This project kicks off with data sourcing from the above dataset. Initially for predicting the sentiment of the review, it is to be preprocessed or cleaned. This follows the following steps.


Downloading the Data
Data Preparation
Uploading to S3
Building the PyTorch Model
Training the Model
Deploying the model for the web app
Using the model for web app

1. Downloading the data
2. Preparing and Processing the Data
3. Uploading the Data to S3
4. Building and Training the PyTorch Model
5. Testing the Model
6. Deploying the model for testing
7. Use the model for testing
8. Deploy the model for the web app
9. Use the model for the web app

- **Downloading the data:**

The IMDB Dataset is downloaded from ai.standford.edu and saved in the data directory.

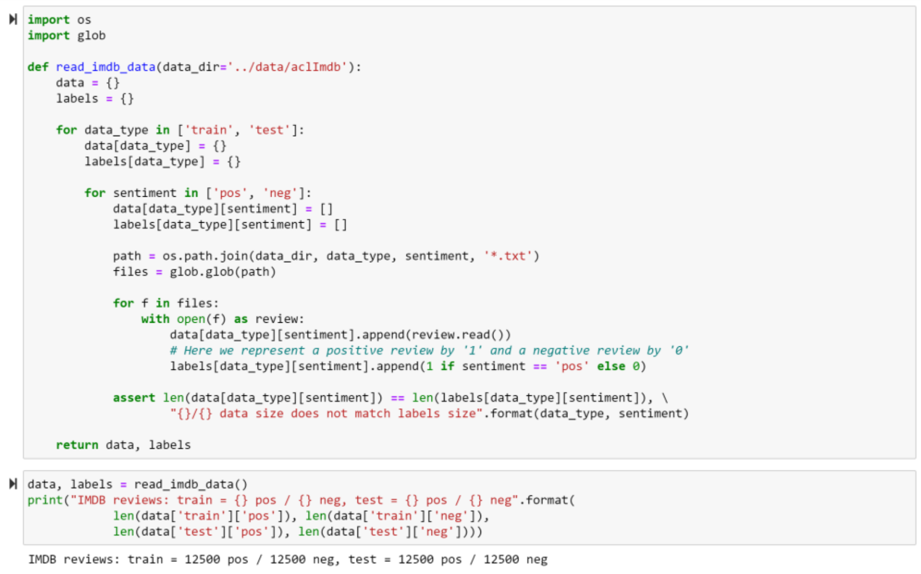


Large Movie Review Dataset

This is a dataset for binary sentiment classification containing substantially more data than previous benchmark datasets. We provide a set of 25,000 highly polar movie reviews for training, and 25,000 for testing. There is additional unlabeled data for use as well. Raw text and already processed bag of words formats are provided. See the README file contained in the release for more details.

## ➢ Preparing and Processing the Data:

The data in the IMDB Dataset is not clean. It is cleaned before sending it to the Machine Learning algorithm. Firstly, the data is split into train and test sets and labels (positive or negative) are given. Once the labels are given to the data, it is shuffled within the train and test data.

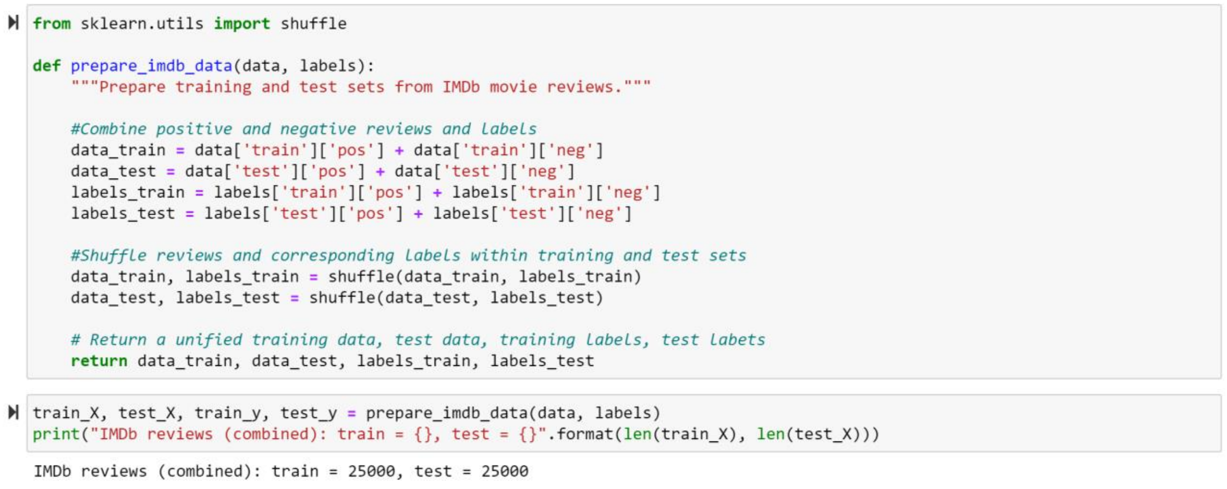


```python
import os
import glob

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 review by '0'
                    labels[data_type][sentiment].append(1 if sentiment == 'pos' else 0)

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

    return data, labels
```

```python
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
```

Fig. Splitting and labelling of train and test data



```python
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
```

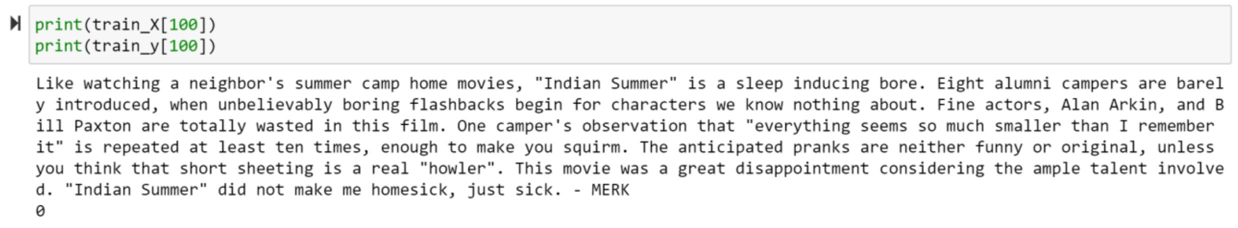
```python
train_X, test_X, train_y, test_y = prepare_imdb_data(data, labels)
print("IMDb reviews (combined): train = {}, test = {}".format(len(train_X), len(test_X)))
```

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

Fig. Shuffling the data

The output indicates 25,000 reviews are given to train data and 25,000 reviews are given to test data.

**Checking the train data:**

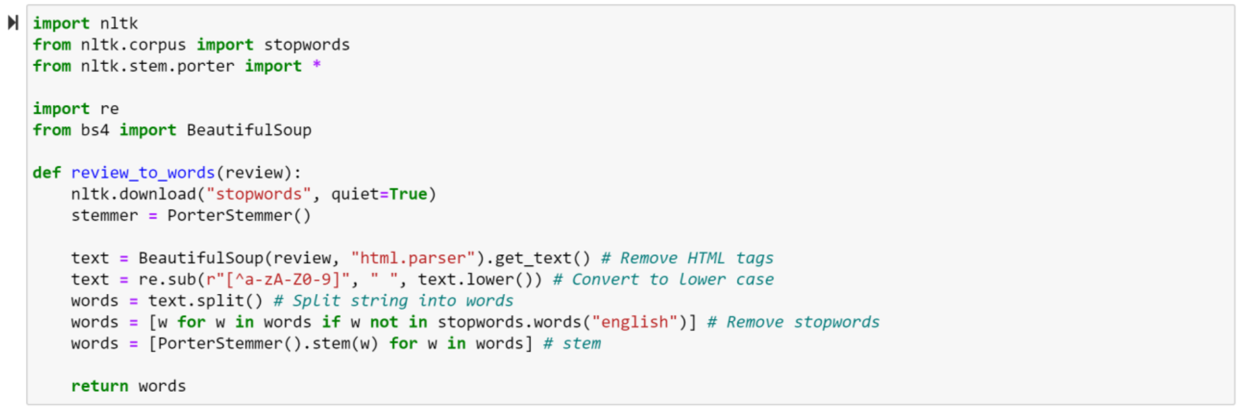


```
print(train_X[100])
print(train_y[100])
```

```
Like watching a neighbor's summer camp home movies, "Indian Summer" is a sleep inducing bore. Eight alumni campers are barel
y introduced, when unbelievably boring flashbacks begin for characters we know nothing about. Fine actors, Alan Arkin, and B
ill Paxton are totally wasted in this film. One camper's observation that "everything seems so much smaller than I remember
it" is repeated at least ten times, enough to make you squirm. The anticipated pranks are neither funny or original, unless
you think that short sheeting is a real "howler". This movie was a great disappointment considering the ample talent involve
d. "Indian Summer" did not make me homesick, just sick. - MERK
0
```

train_X[100] is a review  of a movie. train_y[100] indicates the sentiment of the review which is negative in this case.

The processing is done by removing the html tags in the data, converting all words to lowercase, splitting the text into words and removing the stop words. This is done for every review in the dataset. Finally, the words are tokenized using stemmer. This is done to make sure words such as entertained and entertaining are considered the same with regard to the sentiment analysis.

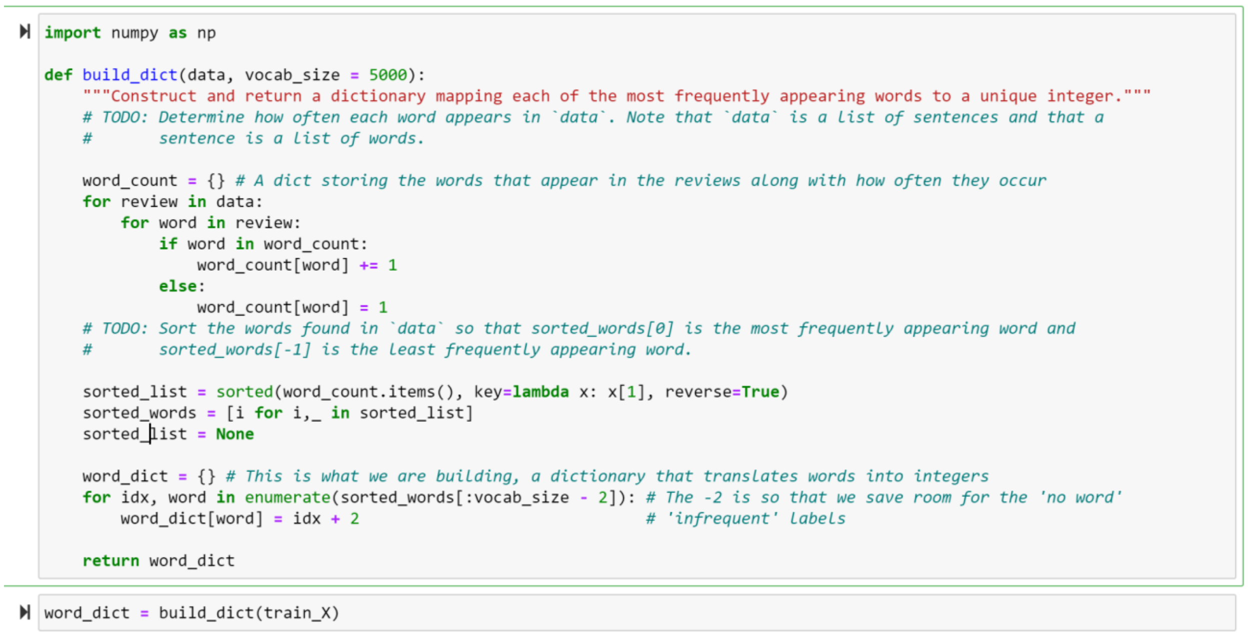


```
import nltk
from nltk.corpus import stopwords
from nltk.stem.porter import *

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 stopwords
    words = [PorterStemmer().stem(w) for w in words] # stem

    return words
```

Fig. review_to_words method for converting the text(review) to words

This processed data is split into train and test data.

Word dictionary or bag of words: For this model, a feature representation is constructed. Each word in the review is represented using an integer. Of course, some of the words that appear in the reviews occur very infrequently and so likely don't contain much information for the purposes of sentiment analysis. This problem is solved by involving only the most frequently occurred words(In our case, first 4998 words) in the reviews and combining all the infrequent words into a single category (label 1).

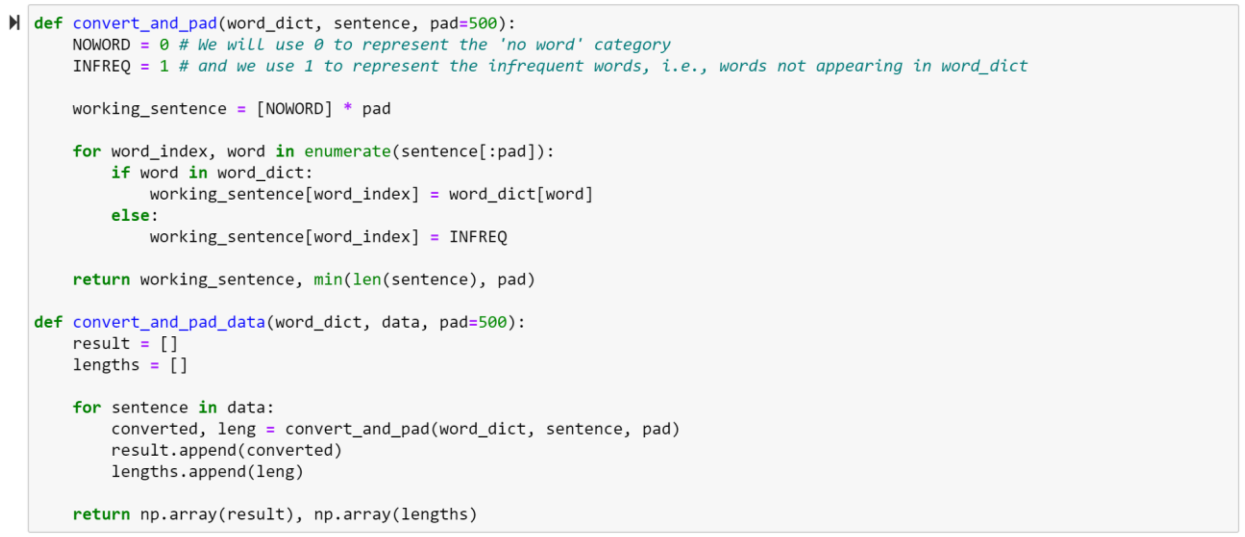


```python
import numpy as np

def build_dict(data, vocab_size = 5000):
    """Construct and return a dictionary mapping each of the most frequently appearing words to a unique integer."""
    # TODO: Determine how often each word appears in `data`. Note that `data` is a list of sentences and that a
    #       sentence is a list of words.

    word_count = {} # A dict storing the words that appear in the reviews along with how often they occur
    for review in data:
        for word in review:
            if word in word_count:
                word_count[word] += 1
            else:
                word_count[word] = 1
    # TODO: Sort the words found in `data` so that sorted_words[0] is the most frequently appearing word and
    #       sorted_words[-1] is the least frequently appearing word.

    sorted_list = sorted(word_count.items(), key=lambda x: x[1], reverse=True)
    sorted_words = [i for i,_ in sorted_list]
    sorted_list = None

    word_dict = {} # This is what we are building, a dictionary that translates words into integers
    for idx, word in enumerate(sorted_words[:vocab_size - 2]): # The -2 is so that we save room for the 'no word'
        word_dict[word] = idx + 2                              # 'infrequent' labels

    return word_dict
```

```python
word_dict = build_dict(train_X)
```

Fig. method to build a dictionary

It is convenient for every review to be of same length as a recurrent neural network (RNN) is used. Review size is fixed to 500 and then short reviews are padded with category (label 0) and truncate long reviews.



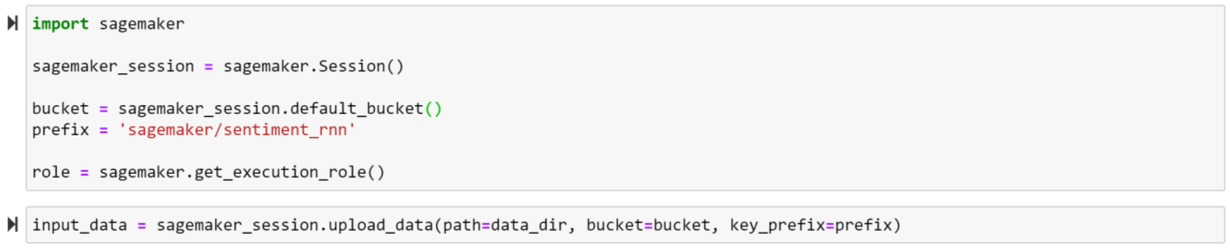
```python
def convert_and_pad(word_dict, sentence, pad=500):
    NOWORD = 0 # We will use 0 to represent the 'no word' category
    INFREQ = 1 # and we use 1 to represent the infrequent words, i.e., words not appearing in word_dict

    working_sentence = [NOWORD] * pad

    for word_index, word in enumerate(sentence[:pad]):
        if word in word_dict:
            working_sentence[word_index] = word_dict[word]
        else:
            working_sentence[word_index] = INFREQ

    return working_sentence, min(len(sentence), pad)

def convert_and_pad_data(word_dict, data, pad=500):
    result = []
    lengths = []

    for sentence in data:
        converted, leng = convert_and_pad(word_dict, sentence, pad)
        result.append(converted)
        lengths.append(leng)

    return np.array(result), np.array(lengths)
```

Fig. method for padding the reviews

This padded data is the one which the algorithm understands. Now all reviews in train and test sets are padded using the word dictionary.

➢ **Uploading the data to S3:**

The training data is uploaded to the S3 bucket by creating a SageMaker session. This is done by creating a default bucket and a prefix which is nothing but a folder in S3 buckets. Now the data is uploaded into the S3 bucket using the upload_data method of sagemaker giving the path to the input data, bucket and the prefix as parameters. In addition to this, a role is created.



```
import sagemaker

sagemaker_session = sagemaker.Session()

bucket = sagemaker_session.default_bucket()
prefix = 'sagemaker/sentiment_rnn'

role = sagemaker.get_execution_role()
```

```
input_data = sagemaker_session.upload_data(path=data_dir, bucket=bucket, key_prefix=prefix)
```

Fig. uploading the data to S3 bucket

## ➢ Building and Training the PyTorch Model:

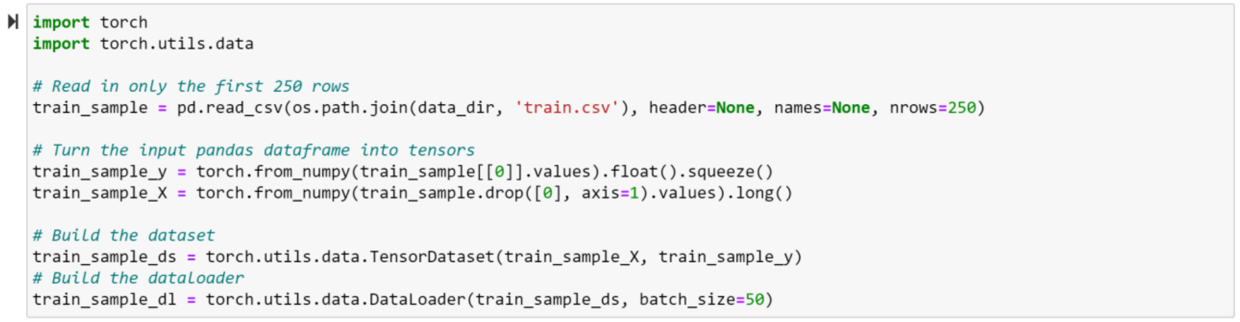
A neural network is implemented in PyTorch along with a training script. To improve the performance of the model, parameters such as embedding dimension, hidden dimension and the vocabulary size are left to be configurable.

```
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())
```

Fig. The Neural Network Model

**<u>Training the model 1</u>:** The data from the directory is loaded into a data frame (only first 250 rows). SageMaker encourages its data to be without a header so the header argument is set to None.

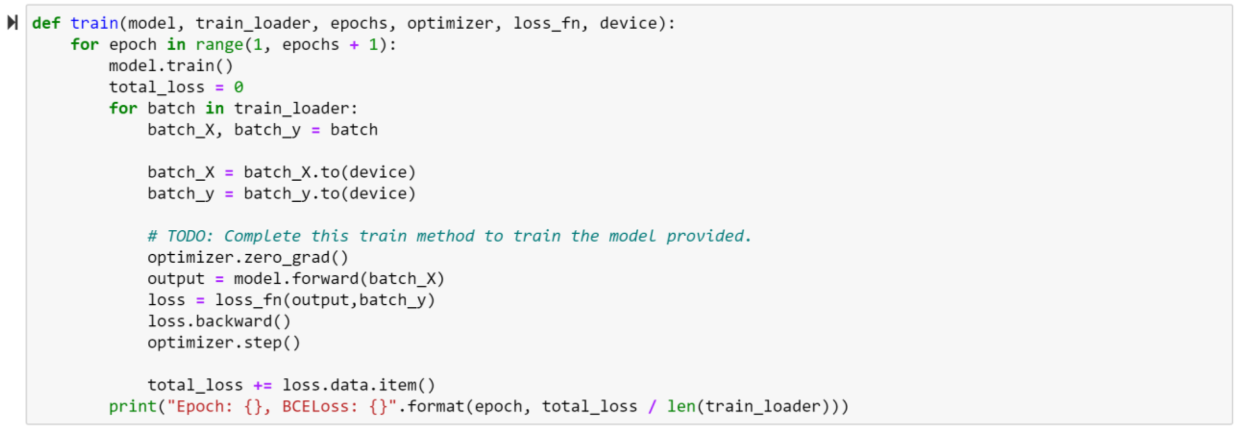
As PyTorch wants its data to be in tensors, the data is converted into tensors. A dataset is built using the tensors. As a neural network excepts the data to be in batches, a data loader is built setting the batch size to 50.



```
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=None, nrows=250)

# 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)
```

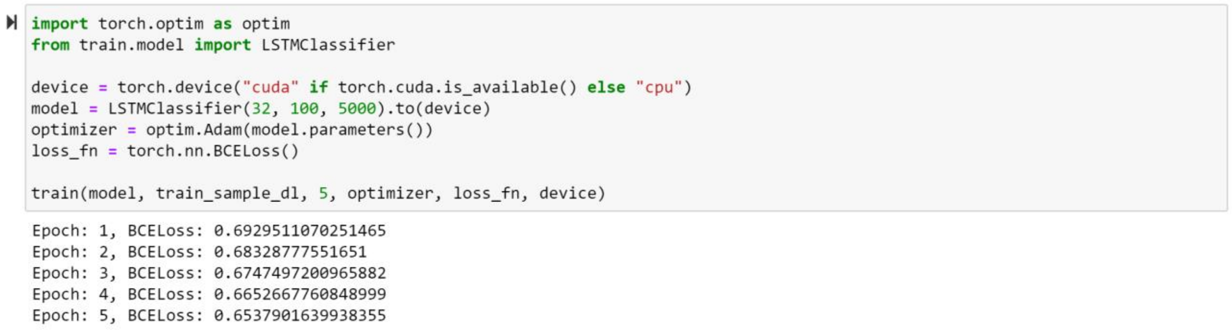
Fig. Sending the data in batches using DataLoader

Now the model is trained. A train function does the training giving the model which was created before, the data in the train loader object, epochs, the optimizer function, loss function, device (whose resources are to be used to train the algorithm) as arguments.



```
def train(model, train_loader, epochs, optimizer, loss_fn, device):
    for epoch in range(1, epochs + 1):
        model.train()
        total_loss = 0
        for batch in train_loader:
            batch_X, batch_y = batch

            batch_X = batch_X.to(device)
            batch_y = batch_y.to(device)

            # TODO: Complete this train method to train the model provided.
            optimizer.zero_grad()
            output = model.forward(batch_X)
            loss = loss_fn(output,batch_y)
            loss.backward()
            optimizer.step()

            total_loss += loss.data.item()
        print("Epoch: {}, BCELoss: {}".format(epoch, total_loss / len(train_loader)))
```

Fig. The train method

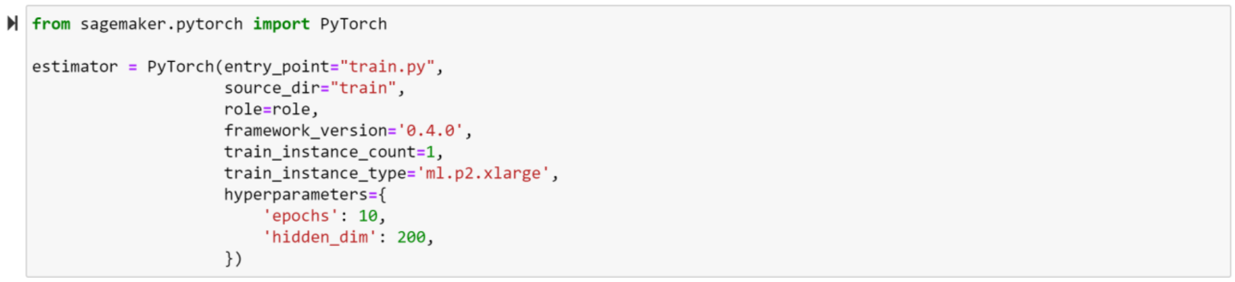


```
import torch.optim as optim
from train.model import LSTMClassifier

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = LSTMClassifier(32, 100, 5000).to(device)
optimizer = optim.Adam(model.parameters())
loss_fn = torch.nn.BCELoss()

train(model, train_sample_dl, 5, optimizer, loss_fn, device)
```

```
Epoch: 1, BCELoss: 0.6929511070251465
Epoch: 2, BCELoss: 0.68328777551651
Epoch: 3, BCELoss: 0.6747497200965882
Epoch: 4, BCELoss: 0.6652667760848999
Epoch: 5, BCELoss: 0.6537901639938355
```

Here, the model used is LSTM Classifier. Optimizer is Adam. Loss Function is BCE.

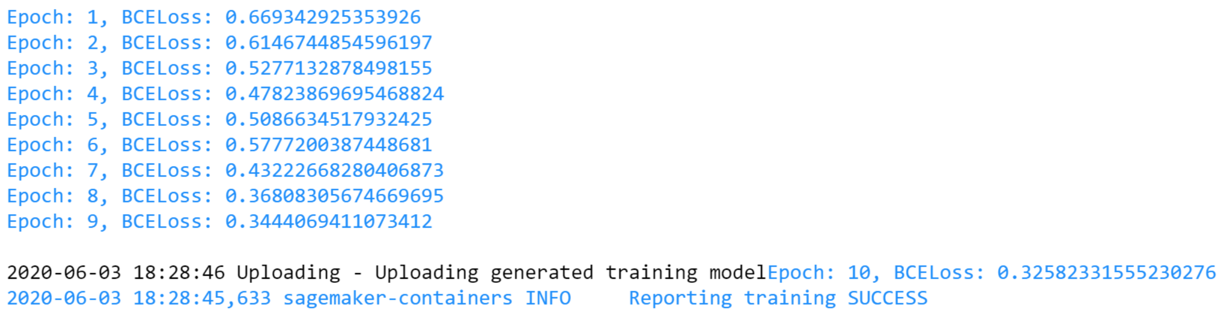
**Training the Model 2:** The model is trained by creating an estimator object of PyTorch. The entry point for the estimator object is the train file in the train directory which is similar to the train function created for Training the Model 1. The role is the execution role created in the beginning. The instance used for training is ml.p2.xlarge and instance count is 1. The hyperparameters epochs, hidden dimensions can be tweaked. In this model, they are set to 10 and 200 respectively.



```
from sagemaker.pytorch import PyTorch

estimator = PyTorch(entry_point="train.py",
                    source_dir="train",
                    role=role,
                    framework_version='0.4.0',
                    train_instance_count=1,
                    train_instance_type='ml.p2.xlarge',
                    hyperparameters={
                        'epochs': 10,
                        'hidden_dim': 200,
                    })
```

Fig. The Estimator Object

The estimator object is fit using the train data (input_data).



```
Epoch: 1, BCELoss: 0.669342925353926
Epoch: 2, BCELoss: 0.6146744854596197
Epoch: 3, BCELoss: 0.5277132878498155
Epoch: 4, BCELoss: 0.47823869695468824
Epoch: 5, BCELoss: 0.5086634517932425
Epoch: 6, BCELoss: 0.5777200387448681
Epoch: 7, BCELoss: 0.43222668280406873
Epoch: 8, BCELoss: 0.36808305674669695
Epoch: 9, BCELoss: 0.3444069411073412

2020-06-03 18:28:46 Uploading - Uploading generated training modelEpoch: 10, BCELoss: 0.32582331555230276
2020-06-03 18:28:45,633 sagemaker-containers INFO     Reporting training SUCCESS
```

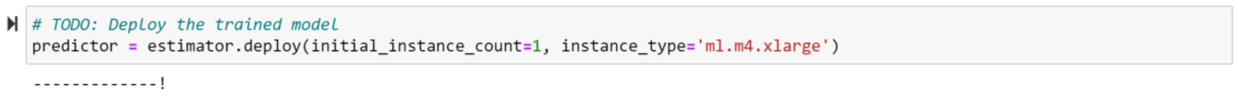
Fig. Output showing the loss at each epoch

**Observation:** The loss is decreasing for every epoch.

- **Deploying the model for testing:**

The trained model is tested by deploying it.

The model is deployed using the estimator object. The instance used for deploying is ml.m4.xlarge and instance count is 1.



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

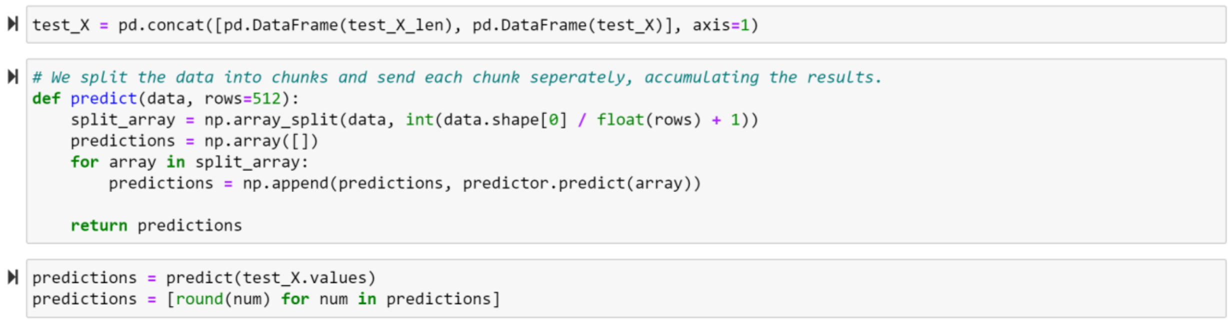
```
-------------!
```

Fig. Deploying the model

- **Using the model for testing:**

After deploying, Model is tested using the endpoint deployed in the previous step.

**Model Testing 1:** The first 512 rows in the test data are used for testing. A predict method is created to predict whether the review is positive or negative using the endpoint.



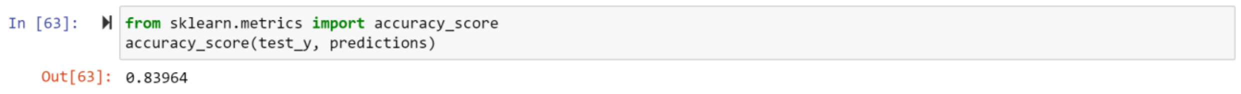
```
test_X = pd.concat([pd.DataFrame(test_X_len), pd.DataFrame(test_X)], axis=1)
```

```
# We split the data into chunks and send each chunk seperately, accumulating the results.
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]
```

Fig. Making predictions using the deployed model

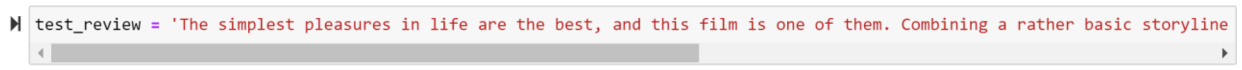
The accuracy score is calculated for the predicted values and the actual values.



```
In [63]:  from sklearn.metrics import accuracy_score
          accuracy_score(test_y, predictions)

Out[63]:  0.83964
```

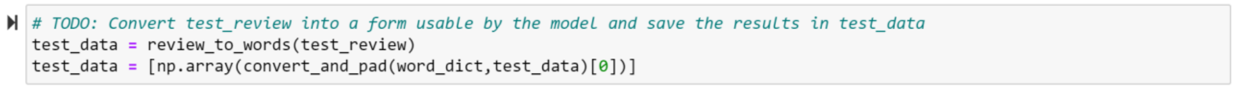
The accuracy of the model is 83% which is decent.

**Model Testing 2:** A sample review is created and stored. The review is processed converting it from text to words and it is tokenized with the help of word dictionary. The review is tested using the endpoint. It returns a float point value. If the value is close to 1, the review is positive else negative.



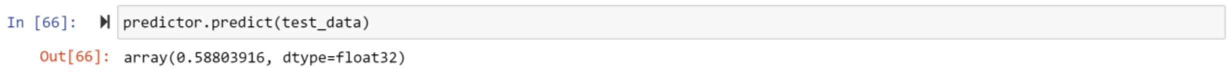
```
test_review = 'The simplest pleasures in life are the best, and this film is one of them. Combining a rather basic storyline
```

Fig. Test review



```
# TODO: Convert test_review into a form usable by the model and save the results in test_data
test_data = review_to_words(test_review)
test_data = [np.array(convert_and_pad(word_dict,test_data)[0])]
```

Fig. Preparing and processing the review



```
In [66]:  predictor.predict(test_data)
Out[66]:  array(0.58803916, dtype=float32)
```

Fig. Predicting the review

As the value is close to 1, the review is positive.

➢ **Deploying the model for web app:**

Its time to create custom inference code so that a review can be sent to the model which is not processed and have it determine the sentiment (positive or negative) of the review.

In the previous steps, the estimator which is used for deployment is provided with a entry script and directory when creating the model. However, now a string input is given but the model expects a processed review. To do this, a custom inference code is needed.

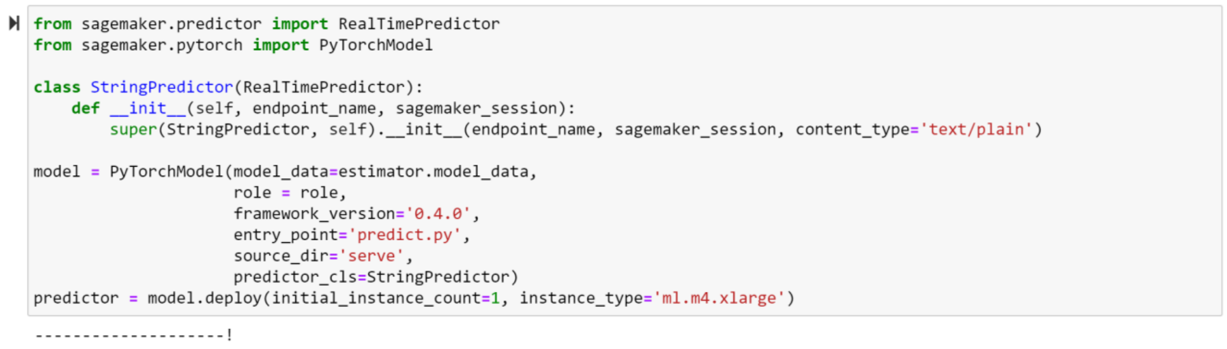
The inference code is in model.py, utils.py and predict.py files. model is a file which is used to construct the model. utils file has the code for processing the data such as tokenizing and padding the data. predict contains the inference code which implements the methods of processing the data from utils.

When deploying a PyTorch model in SageMaker, four functions are expected which the SageMaker inference container will use.

- model_fn: This function is the same function that we used in the training script and it tells SageMaker how to load our model.
- input_fn: This function receives the raw serialized input that has been sent to the model's endpoint and its job is to de-serialize and 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 output and return it to the caller of the model's endpoint.
- predict_fn: The heart of the inference script, this is where the actual prediction is done and is the function which you will need to complete.

For the simple website for this project, the input_fn and output_fn methods are relatively straightforward.They require being able to accept a string as input and expected to return a single value as output.

After writing the inference code, a model is created and deployed. To start with, a new PyTorchModel object which points to the model artifacts creates during training and also points to the inference code. Then deploy method can be called to launch the deployment container.



```
from sagemaker.predictor import RealTimePredictor
from sagemaker.pytorch import PyTorchModel

class StringPredictor(RealTimePredictor):
    def __init__(self, endpoint_name, sagemaker_session):
        super(StringPredictor, self).__init__(endpoint_name, sagemaker_session, content_type='text/plain')

model = PyTorchModel(model_data=estimator.model_data,
                     role = role,
                     framework_version='0.4.0',
                     entry_point='predict.py',
                     source_dir='serve',
                     predictor_cls=StringPredictor)
predictor = model.deploy(initial_instance_count=1, instance_type='ml.m4.xlarge')

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

Fig. PyTorchModel Object and deployment

- **Use the model for the web app:**

Until now the model endpoint is accessed by constructing a predictor object which uses the endpoint and then just the predictor object to perform inference. However, things are different when it comes to a web app accessing the model.

For a web app to access a SageMaker endpoint the app would first have to authenticate with AWS using an IAM role.

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

The magic happens in the middle. A lambda function is constructed, which can be considered as a straightforward python function that can be executed whenever a specified event occurs. Permissions are given to this function to send and receive and receive data from the SageMaker endpoint.

Lastly, an API Gateway is created to send data from the web app to the Lambda Function. The endpoint will be an url that listens for the data to be sent to it. Once it gets some data it will pass that data on to the Lambda function and then return whatever the Lambda function returns. Essentially it will act as an interface that lets our web app communicate with the Lambda function.

## Setting up a Lambda function

Lambda function will be executed whenever the API has data to sent to it. When the lambda function is executed it will receive data (review), performs the preprocessing, send the data to the SageMaker endpoint and then return the result.

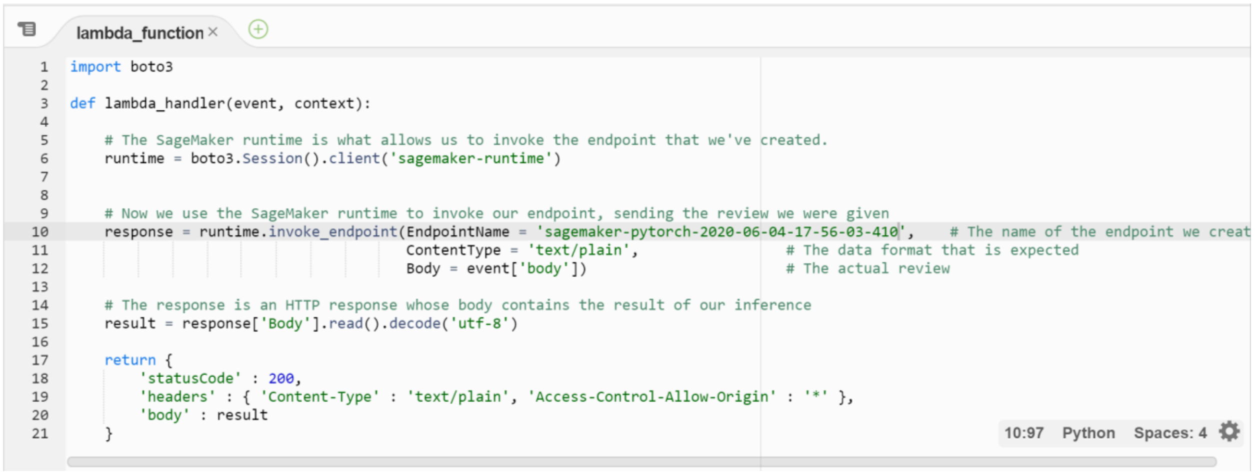
*Part A: Create an IAM Role for the Lambda function*

Lambda function calls the SageMaker endpoint. To do this it needs certain permissions. A role is created which is given to the lambda function later. The permission given to the lambda function in this project is AmazonSageMakerFullAccess.

*Part B: Create a Lambda function*

Now it's time to create the Lambda function. In this project, it is created by 'Author from scratch' and setting the runtime environment to python 3.6. The role created in part A is given to the Lambda function created.

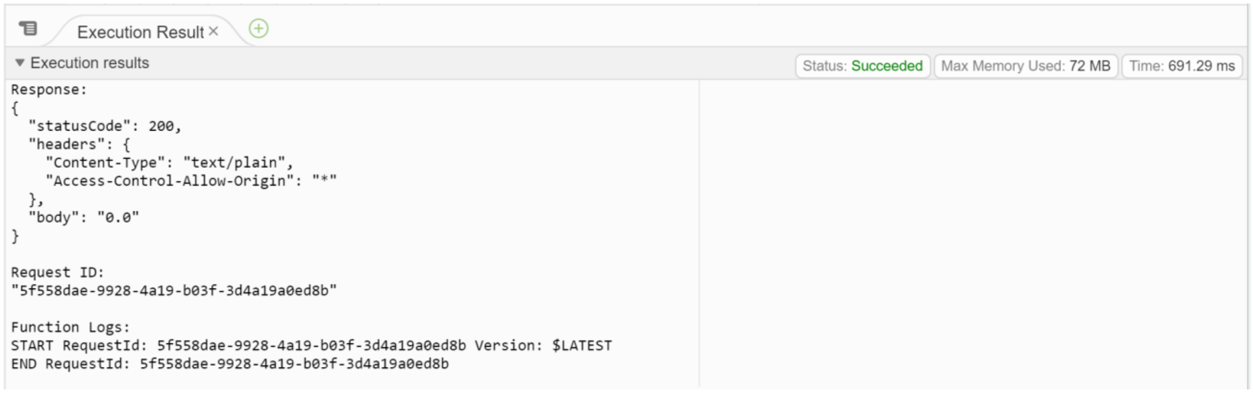
In the editor, lambda function is written mentioning the endpoint of the deployed model.



```python
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 given
    response = runtime.invoke_endpoint(EndpointName = 'sagemaker-pytorch-2020-06-04-17-56-03-410',    # The name of the endpoint we creat
                                       ContentType = 'text/plain',                 # The data format that is expected
                                       Body = event['body'])                       # The actual review

    # 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
    }
```

Fig. The Lambda Function

*Testing the lambda function:*



Status: Succeeded | Max Memory Used: 72 MB | Time: 691.29 ms

```
Response:
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "text/plain",
    "Access-Control-Allow-Origin": "*"
  },
  "body": "0.0"
}

Request ID:
"5f558dae-9928-4a19-b03f-3d4a19a0ed8b"

Function Logs:
START RequestId: 5f558dae-9928-4a19-b03f-3d4a19a0ed8b Version: $LATEST
END RequestId: 5f558dae-9928-4a19-b03f-3d4a19a0ed8b
```

The 0.0 indicate the test review is negative.

**<u>Setting up API Gateway</u>**

An API is needed to trigger the Lambda function created in the previous step. A New REST API is created by giving it a name. To make it trigger the Lambda function created earlier, a POST method is created and selecting Lambda Function as the integration point. By checking Use Lambda Proxy integration, it makes sure the data passing through API doesn't undergo and processing.

Select the Lambda function to be integrated to the API. The last step is to deploy the API. Deploy the API on to a new Deployment stage. In this project, it is prod.

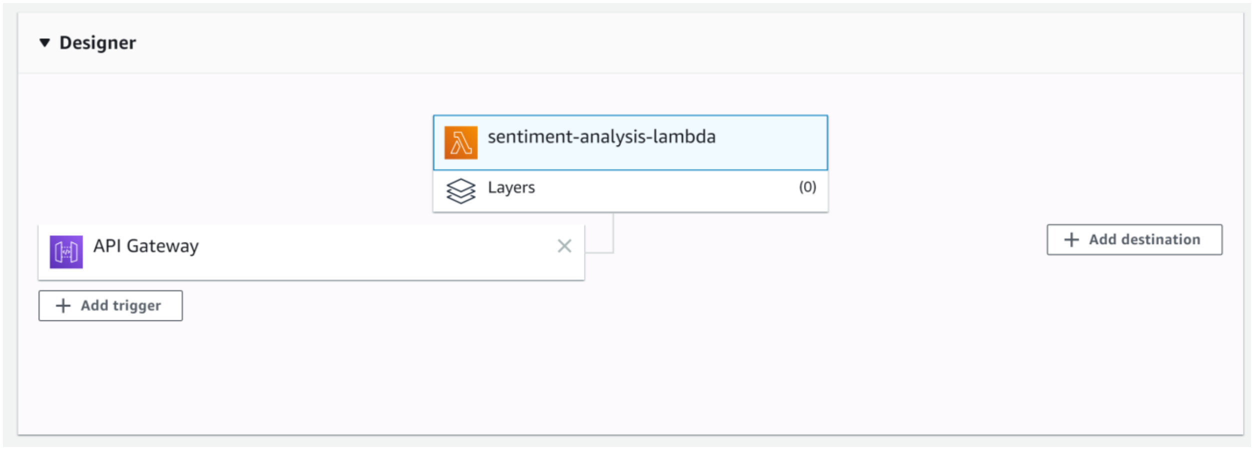
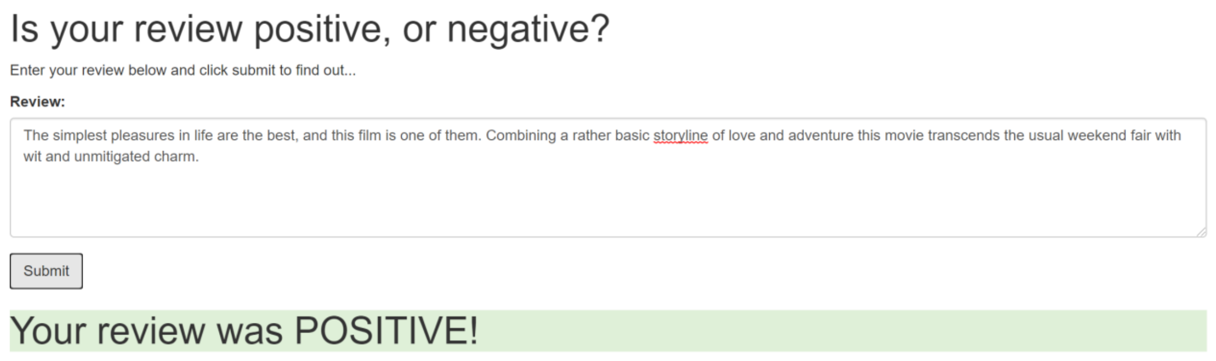


Fig. Integration of Lambda function and public API

A public API is created. Copy the URL provided to invoke the API.

- **Deploying the web app:**

Model deployment facilitates the integration of machine learning model with the production environments.



Is your review positive, or negative?

Enter your review below and click submit to find out...

Review:

The simplest pleasures in life are the best, and this film is one of them. Combining a rather basic storyline of love and adventure this movie transcends the usual weekend fair with wit and unmitigated charm.

Your review was POSITIVE!

Fig. The web app

A sample positive review is given and the output given by the model is positive.

Paste the URL at the appropriate place in the html page for the webpage to pass the information to and from the Lambda Function.