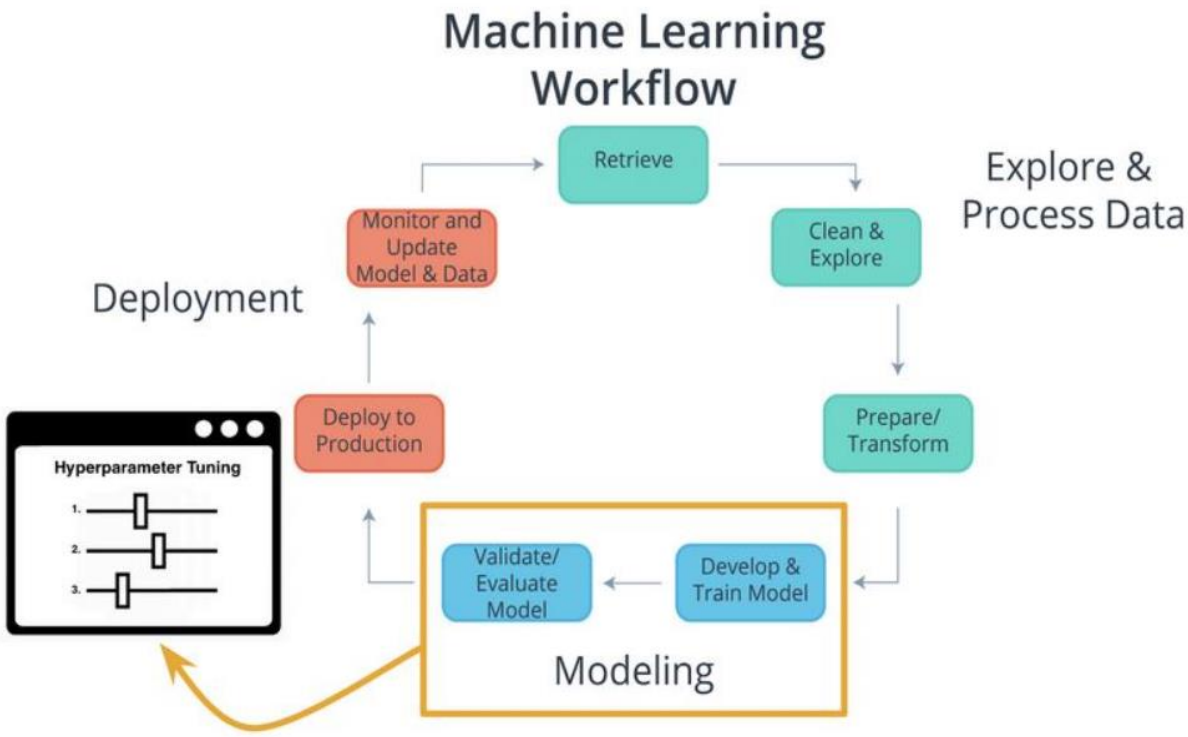
I. Introduction

Human communication is a complex process. People tend to interpret language very differently. Everyone expresses their views, feelings, and opinions differently from other people. That makes it difficult to interpret the sentiment of people's opinion expressed in two, or three sentences. However, this is where Natural Language processing (NLP) and Machine Learning (ML) with Amazon Web Services (AWS) come into play.

The advantages of ML in this case are:

- Data input from unlimited resources
- Fast Processing and Realtime-predictions
- Automation
- Simplification and easy pattern-detection

The goal of this Sentiment analysis will be to set up a Workflow in AWS that takes the written opinion about a movie within a WebApp, processes it, and outputs the sentiment that was expressed in this opinion (was the movie good or bad?).



II. CRISP-DM

The Cross Industry Standard Process for Data Mining (CRISP-DM) is a widely used open standard process model used for Data Mining. It basically consists out the steps listed below and, hence, is used in this Sentiment-Analysis.

1. Business Understanding

Using ML for a Sentiment Analysis is a computing-intense task. This is where AWS comes into play supporting us with computing power in the Cloud. Our main goal is to develop a WebApp where a user could input a review and we automatically receive feedback about the sentiment of the review. Let's see how we could built up such a WebApp.

2. Data Understanding

We are using 50000 of pre-labelled training and testing files, that should train / test our model for it to learn how people express their opinion regarding a movie (a case of supervised learning). After the training the model will use a data-input through a WebApp and return the sentiment of this data input.

```
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.m4.xlarge',
                    hyperparameters={
                        'epochs': 10,
                        'hidden_dim': 200,
                    })
```

```
estimator.fit({'training': input_data})
```

```
t/ml/input","job_name":"sagemaker-pytorch-2021-02-22-17-12-17-337","log_level":20,"model_dir":"/opt/ml/model","modu
le_dir":"s3://sagemaker-us-east-1-703125166535/sagemaker-pytorch-2021-02-22-17-12-17-337/source/sourcedir.tar.gz","
module_name":"train","network_interface_name":"eth0","num_cpus":4,"num_gpus":0,"output_data_dir":"/opt/ml/output/da
ta","output_dir":"/opt/ml/output","output_intermediate_dir":"/opt/ml/output/intermediate","resource_config":{"curre
nt_host":"algo-1","hosts":["algo-1"],"network_interface_name":"eth0"},"user_entry_point":"train.py"}

Invoking script with the following command:

/usr/bin/python -m train --epochs 10 --hidden_dim 200


Using device cpu.
Get train data loader.
Model loaded with embedding_dim 32, hidden_dim 200, vocab_size 5000.

Epoch: 1, BCELoss: 0.6789568176074904
Epoch: 2, BCELoss: 0.641229892263607
Epoch: 3, BCELoss: 0.5334387591906956
Epoch: 4, BCELoss: 0.5202223512591148
```

3. Data Preparation

Taking the reviews as input we first need to split the data into training and test data. Using the training data can show the algorithm some positive and negative (labelled) reviews in order for the algorithm to learn, how to distinguish them.

```
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
```

After we got all the data ready it's time for us to build a model in AWS Sagemaker and finally test it.

```
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
```

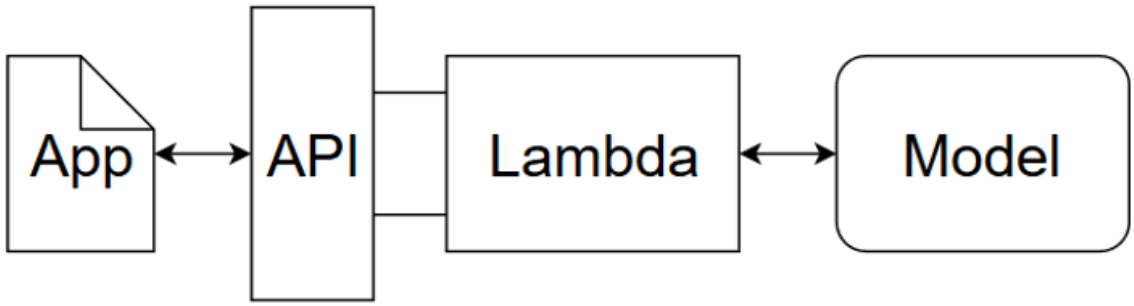
```
predictions = predict(test_X.values)
predictions = [round(num) for num in predictions]
```

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

```
0.82976
```

Testing before developing model

4. Modelling



At this point we've created and deployed a model, and we've constructed a Lambda function that can take care of processing user data, sending it off to our deployed model and returning the result. What we need to do now is set up some way to send our user data to the Lambda function.

In the middle of the above process is where some of the magic happens. We will construct this Lambda function, which you can think of as a straightforward Python function that can be executed whenever a specified event occurs. We will give this function permission to send and receive data from an AWS SageMaker endpoint.

The way that we will do this is using a service called API Gateway. Essentially, API Gateway allows us to create an HTTP endpoint (a web address). In addition, we can set up what we want to happen when someone tries to send data to our constructed endpoint.

In our application, we want to set it up so that when data is sent to our endpoint, we trigger the Lambda function that we created earlier, making sure to send the data to our Lambda function for processing. Then, once the Lambda function has retrieved the inference results from our model, we return the results back to the original caller.

This means you have to type out a review and enter it into the WebApp. After this the WebApp will send that review to an endpoint that we created using API Gateway. API Gateway will forward the data to the Lambda function. The Lambda function is which you can think of as a

straightforward Python function that can be executed whenever a specified event occurs. We will give this function permission to send and receive data from an AWS SageMaker endpoint. It will process that review by tokenizing it and then creating a bag of words encoding of the result. After this it will send the processed review off to our deployed model. Once the deployed model performs inference on the processed review, the resulting sentiment will be sent returned back to the Lambda function. The Lambda function will then return the sentiment results back to the WebApp using the endpoint that was constructed using API Gateway.

5. Evaluation

Before deploying the actual WebApp we need to evaluate our model and check how accurate it predicts the sentiment expressed in the sentences people write into the WebApp. Actually at this stage we should also check the sentiment expressed through the written program on AWS Sagemaker, i.e. checking a sentiment not written in the WebApp, but in the program:

```
import glob

def test_reviews(data_dir='../data/aclImdb', stop=250):

    results = []
    ground = []

    # We make sure to test both positive and negative reviews
    for sentiment in ['pos', 'neg']:

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

        files_read = 0

        print('Starting ', sentiment, ' files')

        # Iterate through the files and send them to the predictor
        for f in files:
            with open(f) as review:
                # First, we store the ground truth (was the review positive or negative)
                if sentiment == 'pos':
                    ground.append(1)
                else:
                    ground.append(0)
                # Read in the review and convert to 'utf-8' for transmission via HTTP
                review_input = review.read().encode('utf-8')
                # Send the review to the predictor and store the results
                results.append(float(predictor.predict(review_input)))

            # Sending reviews to our endpoint one at a time takes a while so we
            # only send a small number of reviews
            files_read += 1
            if files_read == stop:
                break

    return ground, results
```

```
ground, results = test_reviews()
```

```
Starting  pos  files
Starting  neg  files
```

```
from sklearn.metrics import accuracy_score
accuracy_score(ground, results)
```

```
0.834
```

As an additional test, we can try sending the `test_review` that we looked at earlier.

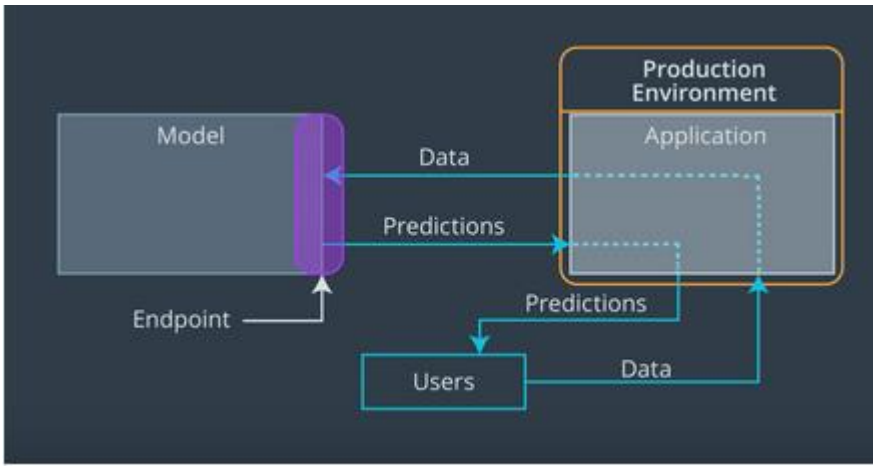
```
predictor.predict(test_review)
```

```
b'1'
```

As you can see we are actually not too bad and receiving a positive feedback (‘1’) for the following test 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.” The result is an output-value of 1 which our WebApp with the help of the lambda-function will later deploy as a positive review.

6. Deployment



Picture: Use the model for the WebApp

The process before launching the WebApp is the following: Build a model > create a lambda function that lets us interact with the model > created an API / endpoint that allows us to send data to lambda function > launch the WebApp.

As seen, the Model has been built up before and we can input our lambda-function. After typing the sentence “The movie was horrible. I do not recommend it at all” the following result occurs. We can see that the result is 0.

Response

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "text/plain",
    "Access-Control-Allow-Origin": "*"
  },
  "body": "0"
}
```

Function Logs

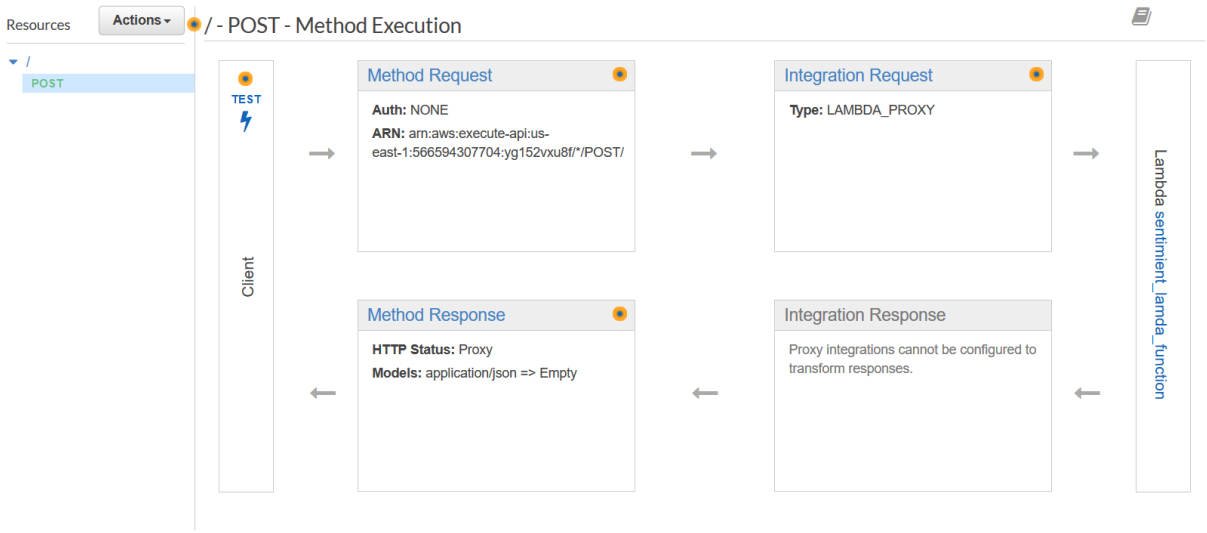
```
START RequestId: 84a07e4d-1f68-472a-a4b7-5210df6e22ae Version: $LATEST
END RequestId: 84a07e4d-1f68-472a-a4b7-5210df6e22ae
REPORT RequestId: 84a07e4d-1f68-472a-a4b7-5210df6e22ae  Duration: 1328.55 ms    Billed Duration: 1329 ms    Memory Size: 128 MB Max Memory Used: 69 MB  Init Duration: 228.24 m
```

Request ID

```
84a07e4d-1f68-472a-a4b7-5210df6e22ae
```

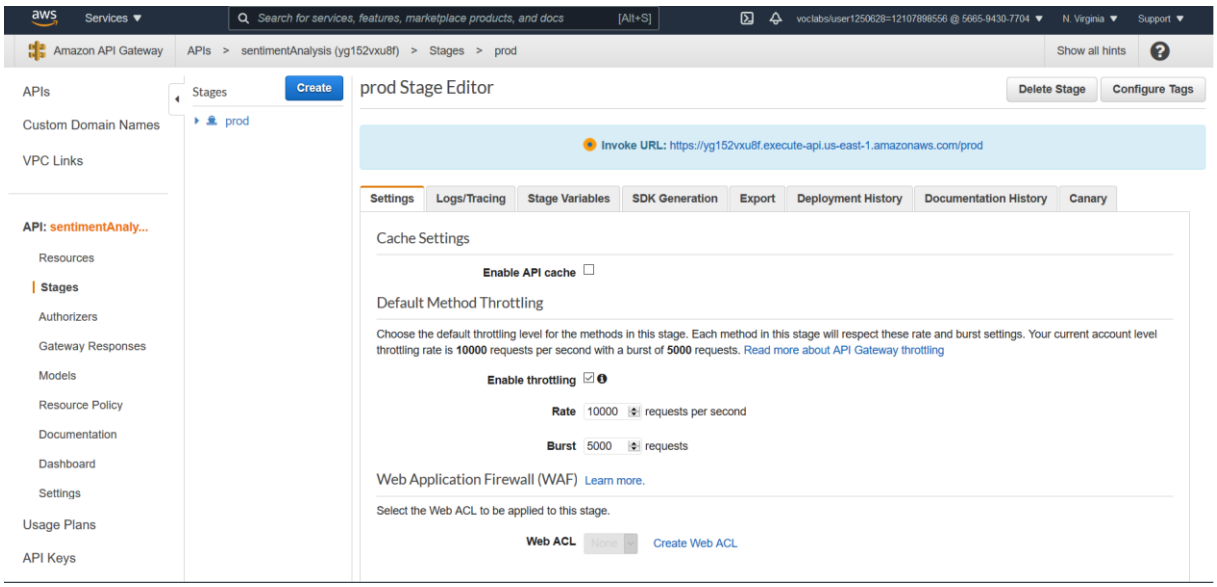
Lambda-function

Now we just need to implement our WebApp and lunch the WebApp. As first step we need to set up an API Gateway that uses our Lambda-function:



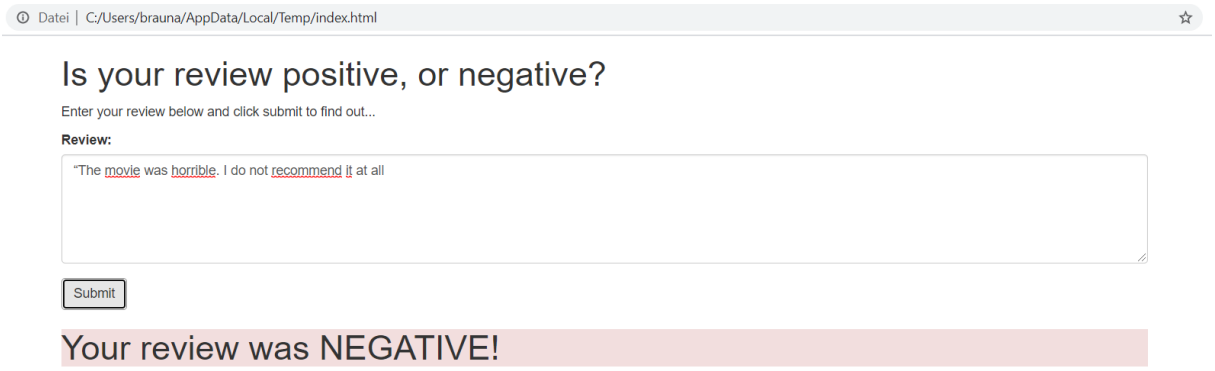
API Gateway

After setting up the API Gateway we need to deploy it to an URL that our API can be accessed with:



Now we have a Lambda function that does the data processing we need, we have a model that creates the data processing we need, and we created an API that allows us to access all of these things. Hence, we just need to lunch the WebApp and see for ourselves how it performs.

After typing the sentence “The movie was horrible. I do not recommend it at all” the following result occurs:



Writing "This was the best movie I have ever seen" brings the following result: