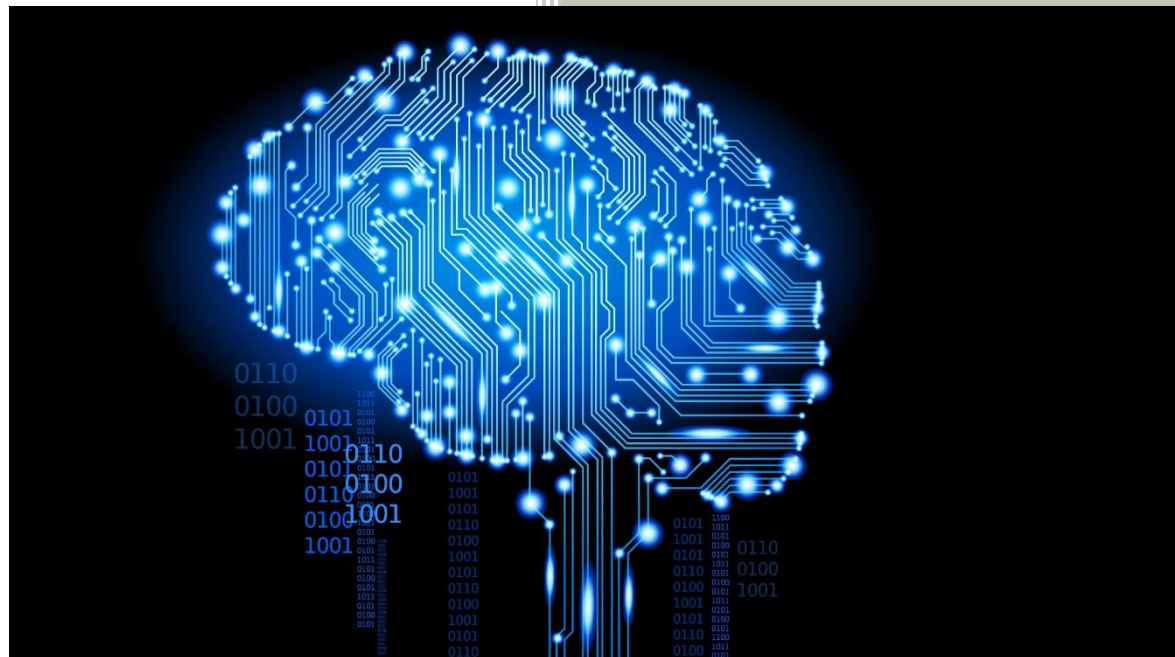


# Documentation for Twitter Sarcasm Detection using BERT



**Submitted by:**

**Saroj Khanal**

**Jiayi Chen**

**Kevin Choi**

**UIUC – Fall 2020**

**CS-410 (Text Information Systems)**

**Course Project**

**Date: 12/12/2020**

## A. Introduction:

As a part of final project of CS 410: Text Information Systems of Fall 2020 from University of Illinois-Urbana Champaign, we formed a team of three members. We decided to go for the Classification competition (<https://github.com/CS410Fall2020/ClassificationCompetition>) where we have to create a model that will classify the given tweets as “Sarcasm” or “Not Sarcasm” and predict the class of 1800 tweets of test set.

After preprocessing, we implemented different methods such as Naïve bayes, LSTM (Long Short-term Memory), BiLSTM (Bidirectional LSTM), and BERT (Bidirectional Encoder Representations from Transformers). Among all, the encoding and the model using BERT gave us the F1 score that was good enough to beat the baseline as required for successful completion of the project. We ran this model over Google Colab and Jupyter Notebook. However, we got the best training time using AWS Sagemaker Notebook instance. The detailed implementation is provided in the presentation document.

## B. Overview of the Code:

The goal of the code is to detect the sarcasm of tweets. The source code for this project performs following task in a sequential order:

- a. Import the training and the test set
- b. Text preprocessing
- c. BERT encoding
- d. Training the model with BERT layer (deep learning)
- e. Use trained model to make prediction on the test set with 1800 tweets (both tweet response and tweet context).

The training set includes 5000 tweets with “Response” and “Context” along with the label as “Sarcasm” or “Not Sarcasm”. It also makes a prediction on the imported twitter test set with output as a text file “answer.txt” that has columns Twitter Id and class label as “Sarcasm” and “Not Sarcasm”. In a nutshell, this code trains on the twitter dataset, classifies whether new tweets are sarcasm or not and it achieves an F1 score over 0.74.

This code can be used for almost any text classification task with some modifications. Even though it only reads JSON file format for now, it can read any kind of acceptable file formats such as `pd.read_csv` and etc. with simple changes.

The code calls specific column names such as Response and Context and they need to be modified as per training set. Even though this code uses two different models, each for Response and Context, one can choose to use single column data with some modifications on the model structure.

Using an activation function such as Softmax instead of Sigmoid can be used to perform a multi-classification instead of a binary classification.

One can modify `max_length` values as needed based on the text length. The lower the value is, the quicker the training process will be, but it is important that this length should be large enough to cover the size of the text (each row) for the optimal result.

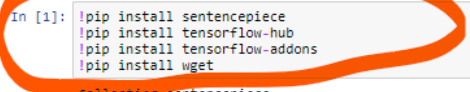
## C. Source Code Implementation:

**How the source code is implemented? How source code is working together to give the prediction file?**

The source code can be split into groups based on the functionality as described below along with the screenshots of source code:

### 1. Installing required libraries and importing preinstalled libraries

This source code requires some libraries other than that comes with the kernel. Here sentencepiece library later helps to receive any iterable object to feed training sentences. Wget helps to download tokenization.py file that will be later used during a tokenization. Tensorflow-hub and tensorflow-addons will be required to run some functions. See below to check the implementation and the output of installing the library. The popular “pip install” is used to download these libraries.



```
In [1]: pip install sentencepiece
pip install tensorflow-hub
pip install tensorflow-addons
pip install wget

Collecting sentencepiece
  Downloading sentencepiece-0.1.94-cp36-cp36m-manylinux2014_x86_64.whl (1.1 MB)
    |#####| 1.1 MB 6.1 MB/s eta 0:00:01
Installing collected packages: sentencepiece
Successfully installed sentencepiece-0.1.94
WARNING: You are using pip version 20.0.2; however, version 20.3.1 is available.
You should consider upgrading via the '/home/ec2-user/anaconda3/envs/tensorflow2_p36/bin/python -m pip install --upgrade pip' c
ommand.
Collecting tensorflow-hub
  Downloading tensorflow_hub-0.10.0-py2.py3-none-any.whl (107 kB)
    |#####| 107 kB 6.2 MB/s eta 0:00:01
Requirement already satisfied: protobuf>=3.8.0 in /home/ec2-user/anaconda3/envs/tensorflow2_p36/lib/python3.6/site-packages (fr
om tensorflow-hub) (3.8.0)
Requirement already satisfied: numpy>=1.12.0 in /home/ec2-user/anaconda3/envs/tensorflow2_p36/lib/python3.6/site-packages (from
tensorflow-hub) (1.18.1)
Requirement already satisfied: six>=1.9 in /home/ec2-user/anaconda3/envs/tensorflow2_p36/lib/python3.6/site-packages (from pr
otobuf>=3.8.0->tensorflow-hub) (1.14.0)
Requirement already satisfied: setuptools in /home/ec2-user/anaconda3/envs/tensorflow2_p36/lib/python3.6/site-packages (from pr
otobuf>=3.8.0->tensorflow-hub) (45.2.0.post20200210)
Installing collected packages: tensorflow-hub
Successfully installed tensorflow-hub-0.10.0
WARNING: You are using pip version 20.0.2; however, version 20.3.1 is available.
You should consider upgrading via the '/home/ec2-user/anaconda3/envs/tensorflow2_p36/bin/python -m pip install --upgrade pip' c
ommand.
Collecting tensorflow-addons
  Downloading tensorflow_addons-0.11.2-cp36-cp36m-manylinux2010_x86_64.whl (1.1 MB)
    |#####| 1.1 MB 6.3 MB/s eta 0:00:01
Collecting typeguard>=2.7
  Downloading typeguard-2.10.0-py3-none-any.whl (16 kB)
Installing collected packages: typeguard, tensorflow-addons
Successfully installed tensorflow-addons-0.11.2 typeguard-2.10.0
WARNING: You are using pip version 20.0.2; however, version 20.3.1 is available.
You should consider upgrading via the '/home/ec2-user/anaconda3/envs/tensorflow2_p36/bin/python -m pip install --upgrade pip' c
ommand.
Collecting wget
  Downloading wget-3.2.zip (10 kB)
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) ... done
  Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=9681 sha256=50728b3c5da93c031b62fb0777826c6fea8dc41b4e51a26c9
1611118bceb29c
  Stored in directory: /home/ec2-user/.cache/pip/wheels/90/1d/93/c863ee832230df5cfc25ca497b3e88e0ee3ea9e44adc46ac62
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
WARNING: You are using pip version 20.0.2; however, version 20.3.1 is available.
You should consider upgrading via the '/home/ec2-user/anaconda3/envs/tensorflow2_p36/bin/python -m pip install --upgrade pip' c
ommand.
```

The picture below includes the list of other libraries that are imported in the notebook. Nltk.punkt and other tokenization tools are used for the purpose of dividing a string into substrings by splitting on the specified string.

```
In [3]: import pandas as pd
import numpy as np
from sklearn.model_selection import StratifiedKFold
import re
import string
from nltk.tokenize import word_tokenize
import nltk
nltk.download('punkt')
import wget
wget.download("https://raw.githubusercontent.com/google-research/ALBERT/master/tokenization.py")

[nltk_data] Downloading package punkt to /home/ec2-user/nltk_data...
[nltk_data] Unzipping tokenizers/punkt.zip.
```

Out[3]: 'tokenization.py'

## 2. Data Preprocessing:

We now perform a data preprocessing. This code has several functions that will work together to preprocess the data. Once the json training and test files are read using Pandas library, the code performs a preprocessing. Some of the notable preprocessing this code performs are:

- a. Remove punctuation
- b. Remove @User
- c. Change abbreviations to normal meaningful strings
- d. Remove emojis
- e. Remove links and non-ASCII characters, if any
- f. Combine separate sentences of context of same tweet into single documents
- g. Change the class label string to numerical forms using labelencoder.

Listed below are some preprocessing functions that we created:

```

In [7]: def clean_tweets(tweet):
        """Removes links and non-ASCII characters"""
        tweet = ''.join([x for x in tweet if x in string.printable])
        # Removing URLs
        tweet = re.sub(r"http\S+", "", tweet)
        return tweet

In [8]: def remove_emoji(text):
        emoji_pattern = re.compile("[
            u"\U0001F600-\U0001F64F" # emoticons
            u"\U0001F300-\U0001F5FF" # symbols & pictographs
            u"\U0001F680-\U0001F6FF" # transport & map symbols
            u"\U0001F1E0-\U0001F1FF" # flags (iOS)
            u"\U00002702-\U000027B0"
            u"\U000024C2-\U0001F251"
        ]+", flags=re.UNICODE)

        return emoji_pattern.sub(r'', text)

In [9]: def remove_punctuations(text):
        punctuations = '@#!?+&*"[]-%.:/(){}$=><|{}^' + "`"
        for p in punctuations:
            text = text.replace(p, f' {p} ')
        text = text.replace('...', ' ... ')
        if '...' not in text:
            text = text.replace('.', ' ... ')
        return text

In [10]: abbreviations = {
        "$" : "dollar ",
        "€" : "euro ",
        "4ao" : "for adults only",
        "a.m" : "before midday",
        "a3" : "anytime anywhere anyplace",
        "aamof" : "as a matter of fact",
        "acct" : "account",
        "adih" : "another day in hell",
        "afaic" : "as far as i am concerned",
        "afaict" : "as far as i can tell",
        "afaik" : "as far as i know",
        "afair" : "as far as i remember",
        "afk" : "away from keyboard",
        "app" : "application",
        "approx" : "approximately",
        "apps" : "applications",
        "asap" : "as soon as possible",
        "asl" : "age, sex, location",

In [11]: def convert_abbrev(word):
        return abbreviations[word.lower()] if word.lower() in abbreviations.keys() else word

In [12]: def convert_abbrev_in_text(text):
        tokens = word_tokenize(text)
        tokens = [convert_abbrev(word) for word in tokens]
        text = ' '.join(tokens)
        return text

In [13]: def preprocessing(df):
        df.response=df.response.str.replace('@USER', "")
        df.response=df.response.str.replace('\d+', '')
        df.response=df.response.str.lower()
        df.response=df.response.str.replace('[^\w\s]','')
        df.context = df.context.apply(lambda x: ', '.join(map(str, x)))
        df.context = df.context.str.replace('@USER', "")
        df.context = df.context.str.lower()
        df.context = df.context.str.replace('[^\w\s]','')
        df.context = df.context.str.replace('\d+', '')
        df.response = df.response.apply(lambda x: clean_tweets(x))
        df.response = df.response.apply(lambda x: remove_emoji(x))
        df.response = df.response.apply(lambda x: remove_punctuations(x))
        df.response = df.response.apply(lambda x: convert_abbrev_in_text(x))
        df.context = df.context.apply(lambda x: clean_tweets(x))
        df.context = df.context.apply(lambda x: remove_emoji(x))
        df.context = df.context.apply(lambda x: remove_punctuations(x))
        df.context = df.context.apply(lambda x: convert_abbrev_in_text(x))
        return df

```

The function `preprocessing` calls other helper functions such as `remove_emoji`, `remove_punctuations`, etc. and applies it to each row of the data using an efficient pandas library

function called “apply”. This is an example of a vectorizing function that works on all rows at the same time and is efficient and faster.

The class labels of the training dataset are in string forms as they are named “Sarcasm” and “Not Sarcasm”. They are converted into numerical forms using LabelEncoder function for the training purpose and saved in as a series with a variable name “train\_label” as shown below:

```
In [20]: #converting Label "Sarcasm" and "Not Sarcasm" to numerical form for training purpose
from sklearn.preprocessing import LabelEncoder
le=LabelEncoder()
train_label=le.fit_transform(twitter_train['label'])
```

We then pass the train and test data frames to this function to get the preprocessed test and train sets as shown below:

```
In [14]: twitter_train = preprocessing(twitter_train)
twitter_test = preprocessing(twitter_test)
```

```
In [15]: twitter_train.head()
```

```
Out[15]:
```

	label	response	context
0	SARCASM	i dont get this obviously you do care or you w...	a minor child deserves privacy and should be k...
1	SARCASM	trying to protest about talking about him and ...	why is he a loser hes just a press secretary h...
2	SARCASM	he makes an insane about of money from the mov...	donald j trump is guilty as charged the eviden...
3	SARCASM	meanwhile trump wont even release his sat scor...	jamie raskin tanked doug collins collins looks...
4	SARCASM	pretty sure the antilincoln crowd claimed that...	man y all gone both sides the apocalypse one d...

```
In [16]: twitter_test.head()
```

```
Out[16]:
```

	id	response	context
0	twitter_1	my year old that just finished reading nietzsc...	well now that s problematic af url my year old...
1	twitter_2	how many verifiable lies has he told now docum...	last week the fake news said that a section of...
2	twitter_3	maybe docs just a scrub of a coach i mean to g...	let s applaud brett when he deserves it he coac...
3	twitter_4	is just a cover up for the real hate inside th...	women generally hate this president whats up w...
4	twitter_5	the irony being that he even has to ask why	dear media remoaners you excitedly sharing cli...

### 3. BERT embedding, Training and Fitting:

After the preprocessing, we now move on to BERT embedding, training, and model fitting. In this step, we conducted the BERT embedding, training and model fitting. First, we downloaded bert\_layer, which was a pre-trained neural network with the Transformer architecture. We chose l=24 as hidden layers. Then, we encoded texts to ids to generate the encoded tokens, masks and segments using the pre-trained bert layers. One thing to be noticed, we encoded response and context separately and combined them afterwards. Finally, we fit the Bert encoded matrices into a model with epochs size of three and batch size of six. We use 90% of the dataset as a training set and 10% as a validation set. As a result, we achieved the F1 score of 0.74 which is about 3% above the baseline.

This is where the initially installed tensorflow\_hub comes into play. This model has been pre-trained for English on the Wikipedia and BooksCorpus using the code published on GitHub. Inputs

have been "uncased", meaning that the text has been lower-cased before tokenization into word pieces, and any accent markers have been stripped.

```
In [21]: import tensorflow_hub as hub
bert_module_url = "https://tfhub.dev/tensorflow/bert_en_uncased_L-24_H-1024_A-16/1" #
bert_layer = hub.KerasLayer(bert_module_url, trainable=True)
```

The following function converts the token into an encoding that is later used as an input to a Bert layer (neural network).

```
In [22]: def bert_encode(texts, tokenizer, max_len=512):
    all_tokens = []
    all_masks = []
    all_segments = []

    for text in texts:
        text = tokenizer.tokenize(text)

        text = text[:max_len-2]
        input_sequence = ["[CLS]"] + text + ["[SEP]"]
        pad_len = max_len - len(input_sequence)

        tokens = tokenizer.convert_tokens_to_ids(input_sequence)
        tokens += [0] * pad_len
        pad_masks = [1] * len(input_sequence) + [0] * pad_len
        segment_ids = [0] * max_len

        all_tokens.append(tokens)
        all_masks.append(pad_masks)
        all_segments.append(segment_ids)

    return np.array(all_tokens), np.array(all_masks), np.array(all_segments)
```

The following set of code tokenizes the sentences and embeds them using BERT. Here we chose max\_len of 256. Choosing the right number was part of tuning the model since the smaller number will speed up the training process in expense of the performance of the model. This number of 256 works well for us and we got the best speed and performance with it.

```
In [24]: import tokenization
vocab_file = bert_layer.resolved_object.vocab_file.asset_path.numpy()
do_lower_case = bert_layer.resolved_object.do_lower_case.numpy()
tokenizer = tokenization.FullTokenizer(vocab_file, do_lower_case)
```

```
In [25]: max_len=256
```

```
In [26]: train_response = bert_encode(twitter_train.response, tokenizer, max_len=max_len)
train_context = bert_encode(twitter_train.context, tokenizer, max_len=max_len)
train_labels=twitter_train.label.values
```

```
In [27]: test_response = bert_encode(twitter_test.response, tokenizer, max_len=max_len)
test_context = bert_encode(twitter_test.context, tokenizer, max_len=max_len)
```

The following code creates a neural network model with the context and the response as inputs. However, we encoded the response and the context separately, and created separate neural network models and later concatenated them together right before they went to the output layer.



Lists `test_generate` and `train_generate` are the output of BERT encoding and they are required to feed into a bert layer of the neural network.

We use the sigmoid function as we have a binary output. We set the learning rate to  $1e-6$  after a few optimizations runs and use Adam as an optimizer since it is one of the most popular optimizers in the industry nowadays. Adam moves faster at first and then slows down once it starts to get closer to the local/global minima while training.

#### 4. Making prediction based on the trained model

Once we create a model, we train it with our training set. We have used 90% of training set as the training data and 10% as the validation dataset. The values for the parameters Training set percentage, batch size and epoch were empirically determined by us to get the best f1 score and the shown values gave us the best result.

Note that each epoch took about one and a half hours even after running with Sagemaker which has the larger RAM than the other options. Running them on a local computer would take almost a day to run 2-3 epoch and it is one of the reasons why we chose Sagemaker as a platform to train our model.

```
In [33]: train_hist = model.fit(
          train_generate, train_label,
          epochs=3,
          batch_size=6,
          validation_split=0.1)

Train on 4500 samples, validate on 500 samples
Epoch 1/3
4500/4500 [=====] - 4724s 1s/sample - loss: 0.5347 - accuracy: 0.7338 - val_loss: 0.8505 - val_accuracy: 0.4980
Epoch 2/3
4500/4500 [=====] - 4622s 1s/sample - loss: 0.3965 - accuracy: 0.8218 - val_loss: 0.9566 - val_accuracy: 0.4680
Epoch 3/3
4500/4500 [=====] - 4622s 1s/sample - loss: 0.3074 - accuracy: 0.8749 - val_loss: 0.7953 - val_accuracy: 0.5760
```

Once the training is completed, we use our test set to make a prediction. Once the prediction is made, it is converted back to non-numerical form of class labels as “Sarcasm”, “Non-Sarcasm” using `LabelEncoder` and `inverse_transform`. The prediction is later to be converted into Dataframe, and concatenated with the twitter ids from the test set so we can have dataframe with twitter ids and a prediction class. The dataframe is later saved as a text file “answer.txt” which is later uploaded on github.

```
In [34]: test_fit=model.predict(test_generate, batch_size=6, verbose=1).ravel()

1800/1800 [=====] - 614s 341ms/sample

In [35]: result=test_fit.round()
          result=le.inverse_transform(result.ravel().astype('int16')) #inverse transform the Label back to "Sarcasm" and "Not Sarcasm"
          result=pd.DataFrame(result,columns=['label'])
          result=pd.concat([twitter_test['id'], result], axis=1)
          result.head()
          result.to_csv('./answer.txt',sep=',',index=False,header=None) #saving the prediction as text file with name "answer.txt"

In [ ]:
```



Since github is already set with the webhook, once we commit and push, the result of our prediction will show on livelab. At the time of writing this documentation we got the F1 score of 0.742 and we were ranked at 30 in the leaderboard.

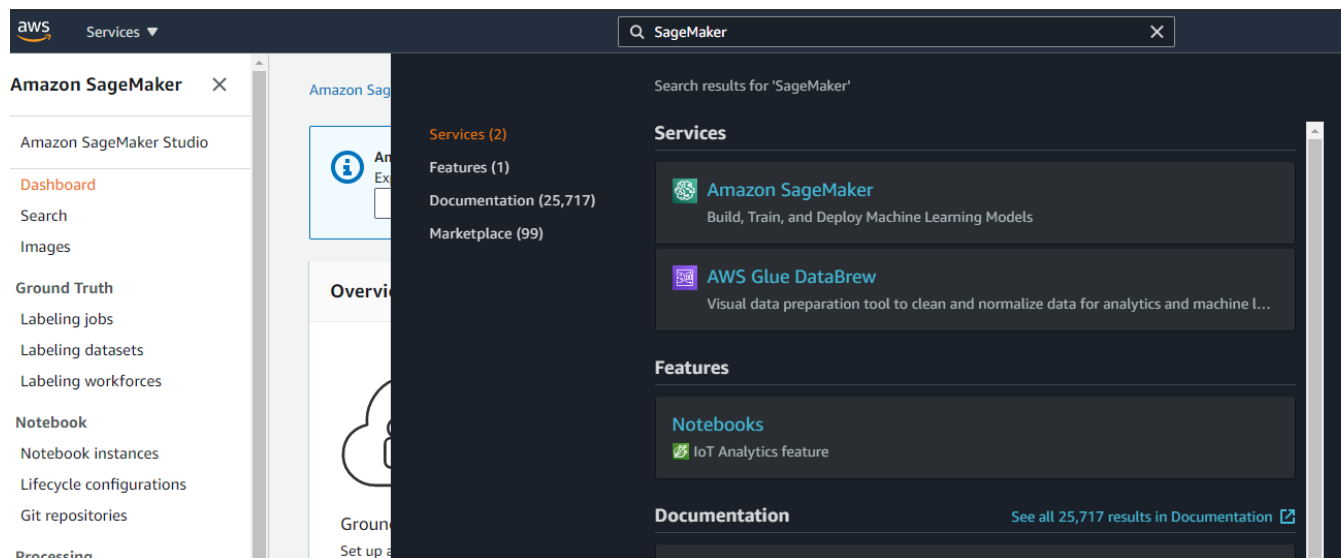
30	khanal2	11	0.6486710963455149	0.8677777777777778	0.7423954372623575	1
----	---------	----	--------------------	--------------------	--------------------	---

#### D. How to run the Source Code?

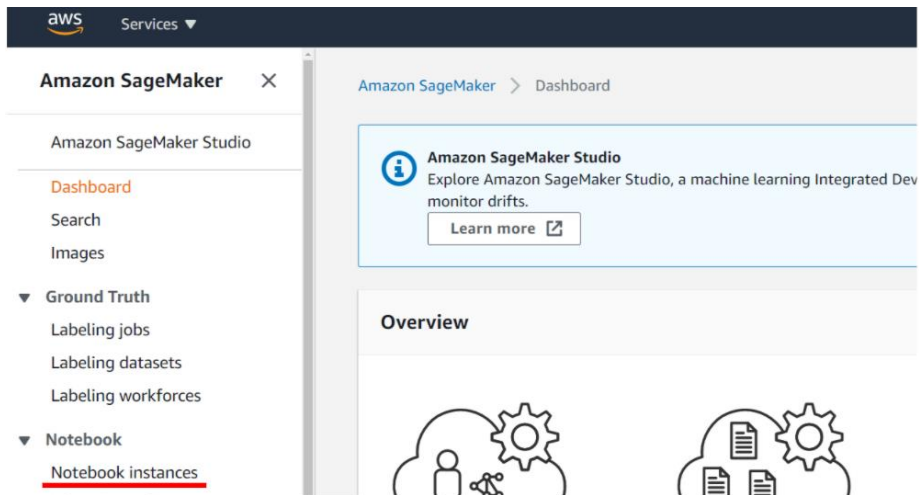
The code will install all the necessary libraries, import all required libraries, therefore there is no need for any additional installation to run this code provided that this code is ran under specific kernel of “**conda\_amazonei\_tensorflow2\_p36**” which is available in Notebook instance of AWS Sagemaker.

Here are the stepwise details on how to run the source code (notebook) on AWS Sagemaker.

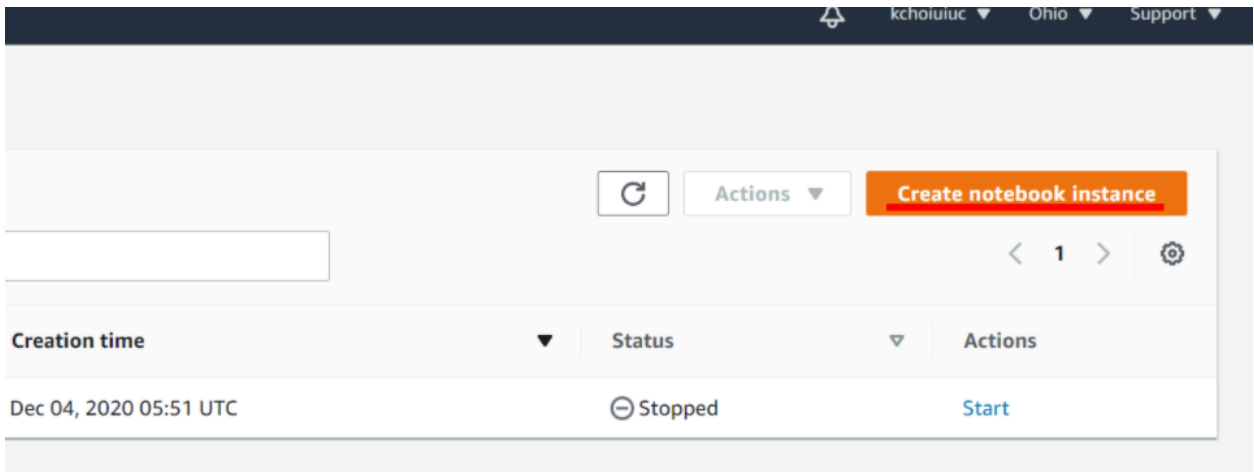
1. Go to <https://aws.amazon.com/> and create an account if you do not have one.
2. Once you are logged in, type SageMaker on the search tab.



3. Go on Notebook Instances as shown below.



4. Click on Create Notebook Instance



5. Type any name for Notebook Instance name and select Instance type. The free tier AWS version only allows certain maximum size. We use ml.c4.8xlarge to train our model which is available for a free tier account.

Amazon SageMaker > Notebook instances > Create notebook instance

## Create notebook instance

Amazon SageMaker provides pre-built fully managed notebook instances that run Jupyter notebooks. The notebook instances include example code for common model training and hosting exercises. [Learn more](#)

### Notebook instance settings

Notebook instance name

Maximum of 63 alphanumeric characters. Can include hyphens (-), but not spaces. Must be unique within your account in an AWS Region.

Notebook instance type

Maximum memory size for a free tier account (YMMV)

Elastic Inference [Learn more](#)

► Additional configuration

- Once you create a notebook instance, it will take 1-2 minutes to be activated (inService). Once you see an inService sign, you can click "Open Jupyter" as below. It will open a notebook on your default browser.

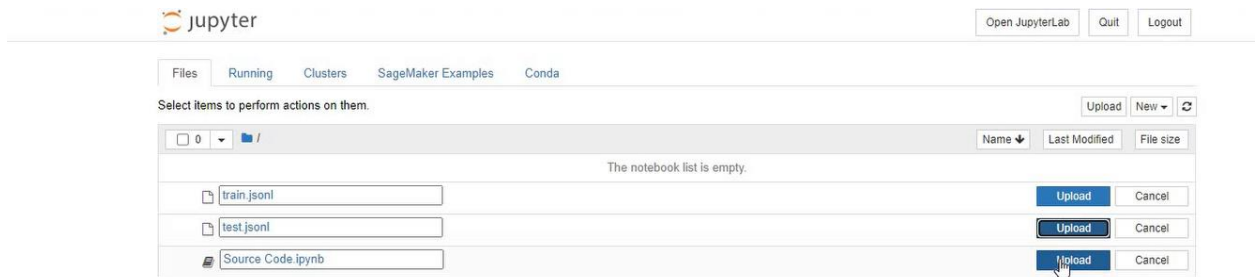
kchoiuiuc
Ohio
Support

Delete
Stop
Open Jupyter
Open JupyterLab

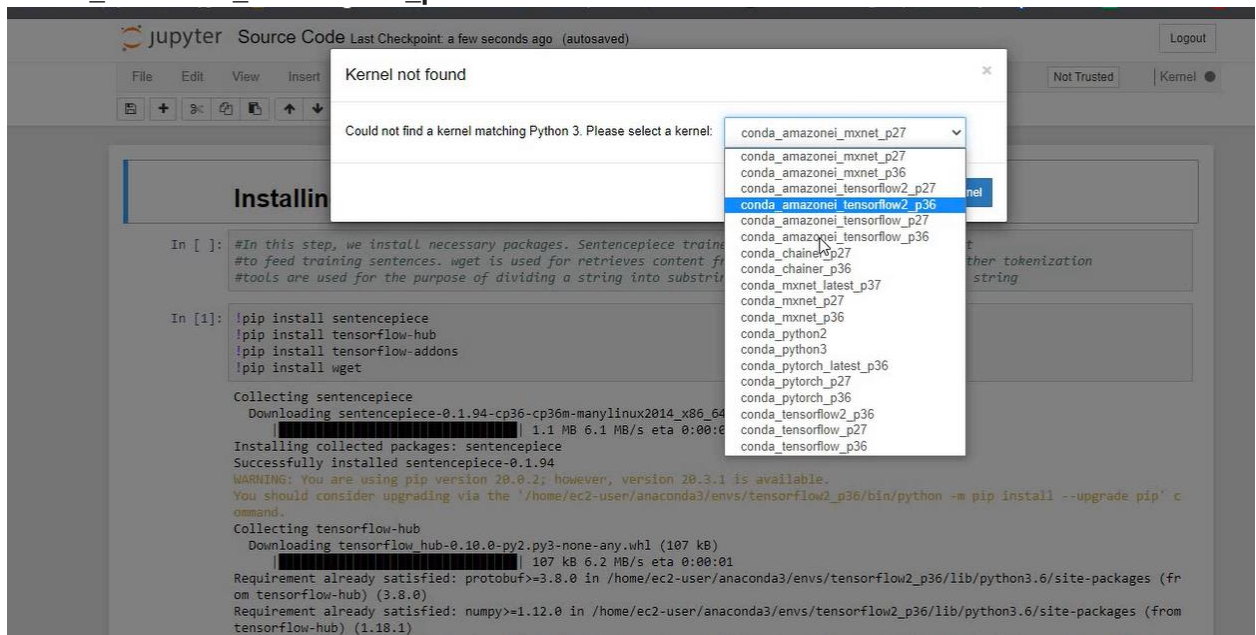
Edit

Status	Notebook instance type
✓ InService	ml.c4.8xlarge
Creation time	Elastic Inference
Dec 04, 2020 05:51 UTC	-
Last updated	Volume Size
Dec 11, 2020 23:39 UTC	5GB EBS

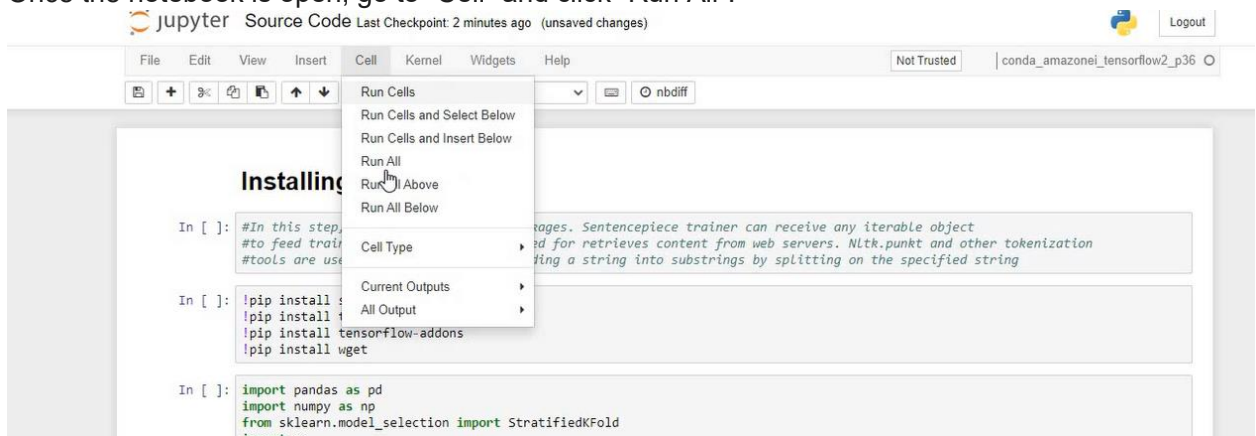
- Once your AWS notebook opens, upload the source code (notebook file), train.jsonl, and test.jsonl file on the notebook.



8. Once your upload is finished, click on notebook ("Source code").
9. Change the kernel to **conda\_amazonei\_tensorflow2\_p36** by selecting the kernel tab from the navigation menubar and select "change kernel" tab to choose **conda\_amazonei\_tensorflow2\_p36** kernel.



10. Once the notebook is open, go to "Cell" and click "Run All".



All the code cell will run and the prediction on test set will be exported as “answer.txt” after a few hours on the root directory that can be accessed by going File and click “Open”. This txt file will have the prediction as per the project requirement.

### How did we get here? What are our experiments with other methods and hyperparameter tuning?

Once we did the preprocessing, we used different machine learning algorithm to make predictions. While other ML algorithm was not giving us a good result, BiLSTM was giving us some encouraging result. Below are the screenshots of one of the models we used. This one gave us the F1 score of about 0.65 which was not good enough. After some suggestion from the discussion board, we came to realize that BERT with its attention layer will definitely help us to achieve the prediction on test set that could beat the baseline.

```
In [24]: ## Creating model
embedding_vector_features=40
model1=Sequential()
model1.add(Embedding(voc_size,embedding_vector_features,input_length=sent_length))
model1.add(Bidirectional(LSTM(100)))
model1.add(Dropout(0.3))
model1.add(Dense(1,activation='sigmoid'))
model1.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])
print(model1.summary())
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 730, 40)	200000
bidirectional (Bidirectional)	(None, 200)	112800
dropout (Dropout)	(None, 200)	0
dense (Dense)	(None, 1)	201

Total params: 313,001  
 Trainable params: 313,001  
 Non-trainable params: 0

All three of us started our own research and share ideas and references with each other. We came up with one model that started to look promising with the F1 score of 0.69. That is the time when we realized that all we need was a good hyperparameter tuning.

Here are the lists of parameters that we tuned in order to get the result to beat the baseline.

- Epoch
- Learning batch size
- Learning rate
- Max Length to feed into Bert Layer (max\_length)
- Test/Validation split percentage

Since the training took us over 4 hours for a single epoch with 15 batch size, we needed to find a way to speed up the process. That is when we decided to go with Sagemaker where we could leverage larger RAM size (60 gb RAM with 36vCPU). After reducing the training time from 4 hours to 40 mins using Sagemaker, we were able to run the model with different

hyperparameters more easily. Our `max_length` at first was 150 but later on we changed it to 256 in order to get the better result.

We were able to get closer to baseline after a few trials, but we still could not beat the baseline by a small margin. We then went back and started making changes on the preprocessing. Realizing the feature engineering was a very important aspect of the training, we made some changes such as converting abbreviated words to natural words and implementing the better handling of punctuations by keeping some punctuations such as (...). With these changes and a few more trials with the different hyperparameters, we managed to beat the baseline.

### **How was our team effort? Who did what?**

After our initial meetings, we decided that we all should be working separately during the preprocessing. There were two main reason for this decision:

- a. It was necessary to come up with the preprocessed data since we cannot move forward without it. Therefore, coming up with a near-perfect preprocessing was important.
- b. Working independently would bring more creativity while preprocessing.

Once we came up with our own set of the preprocessing, we had a meeting and come up with the best preprocessing code which included best parts of all three different preprocessing code.

We then decided to work on different models and compare results. Saroj worked on BiLSTM and `k_train` library, Jiayi worked on BERT encoding and layer, while Kevin worked on Naïve Bayes, and an additional feature engineering that could be possible to implement. While BiLSTM showed some good signs, BERT started to get closer to the baseline. At that point, we all started working on BERT and tried different hyperparameters to train the models. We have each tried over 15 different trainings with different hyperparameters and it took about three hours per training on average. Sometimes we ran two different models simultaneously leveraging AWS Sagemaker. Once we beat the baseline, Jiayi worked on cleaning and organizing the source code, Kevin worked on the presentation and Saroj worked on the first draft of the documentation. Later, everybody came together to finalize the source code, presentation, and demo video.

It was a good team effort, and we all contributed almost the same amount of time. With the training time accounted, we worked over 35 hours each for this project.

### **References:**

- [https://tfhub.dev/tensorflow/bert\\_en\\_uncased\\_L-24\\_H-1024\\_A-16/1](https://tfhub.dev/tensorflow/bert_en_uncased_L-24_H-1024_A-16/1)
- <https://www.kaggle.com/rftexas/text-only-kfold-bert>
- <https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270>
- [https://huggingface.co/transformers/model\\_doc/bert.html](https://huggingface.co/transformers/model_doc/bert.html)
- <https://www.kaggle.com/funxexcel/keras-bert-using-tfhub-trial>