

**Submitted by:**

**Saroj Khanal**

**Jiayi Chen**

**Kevin Choi**

**UIUC – Fall 2020**

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

**Course Project**

**Date: 12/12/2020**

1. **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.

1. **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:

1. Import the training and the test set
2. Text preprocessing
3. BERT encoding
4. Training the model with BERT layer (deep learning)
5. 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.

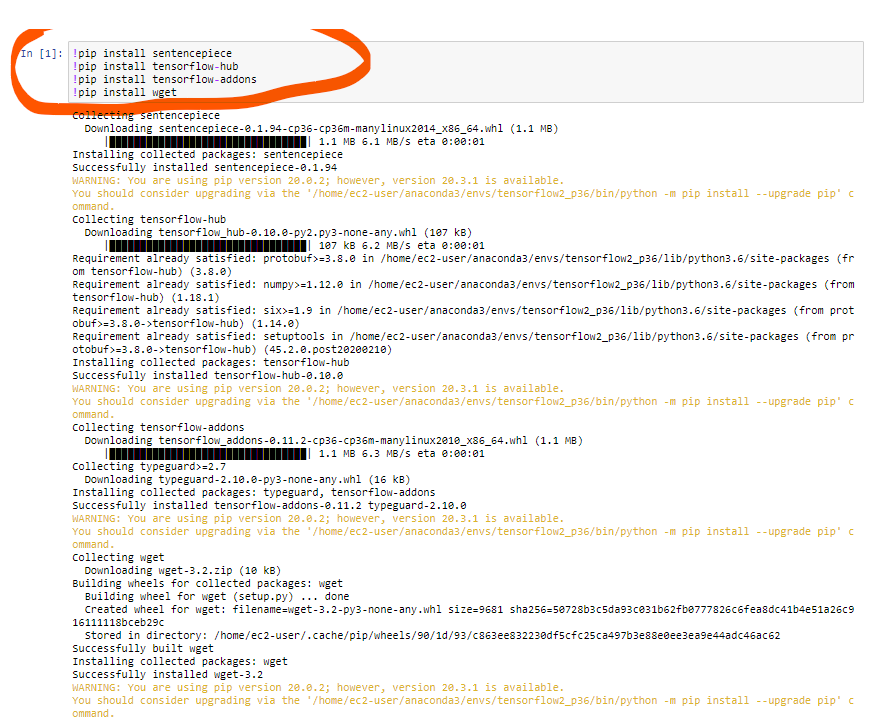
1. **Source Code Implementation:**

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

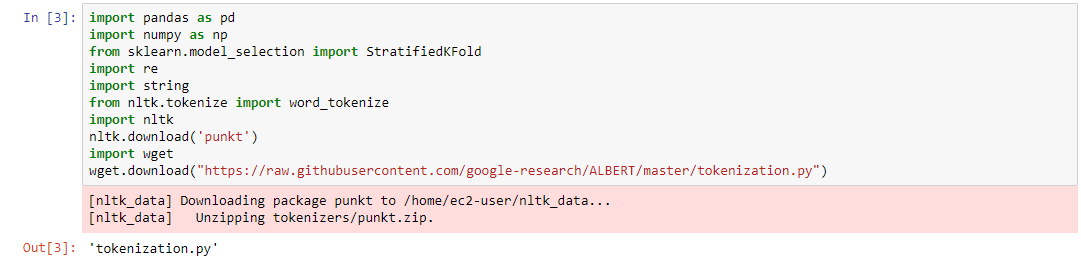
The source code can be split into groups based on the functionality as described below along with the screenshots of source code:

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.



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.



1. **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:

1. Remove punctuation
2. Remove @User
3. Change abbreviations to normal meaningful strings
4. Remove emojis
5. Remove links and non-ASCII characters, if any
6. Combine separate sentences of context of same tweet into single documents
7. Change the class label string to numerical forms using labelencoder.

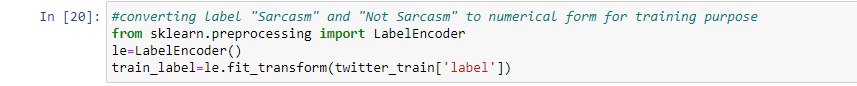
Listed below are some preprocessing functions that we created:



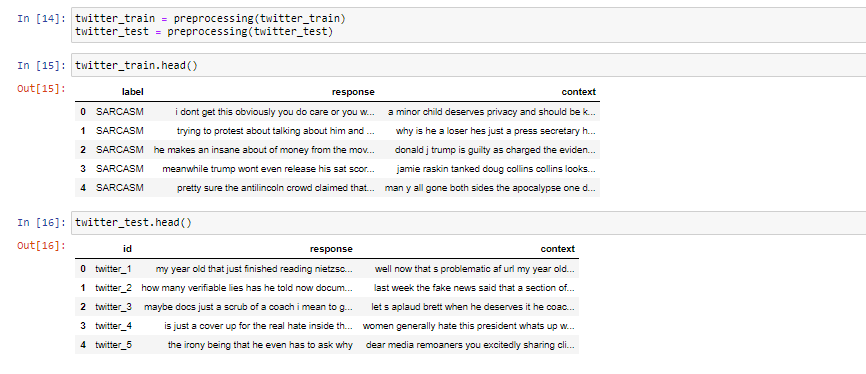


The function preprocessing calls other helper functions such as remove\_emoji, remove\_punctuations, etc. and applies it to each row of the data using an efficient pandas library function called “apply”. This is an example of a vectorizing function that works on all rows at the same time and is efficient 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:



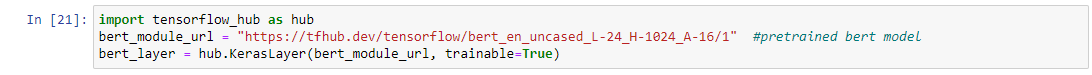
We then pass the train and test data frames to this function to get the preprocessed test and train sets as shown below:



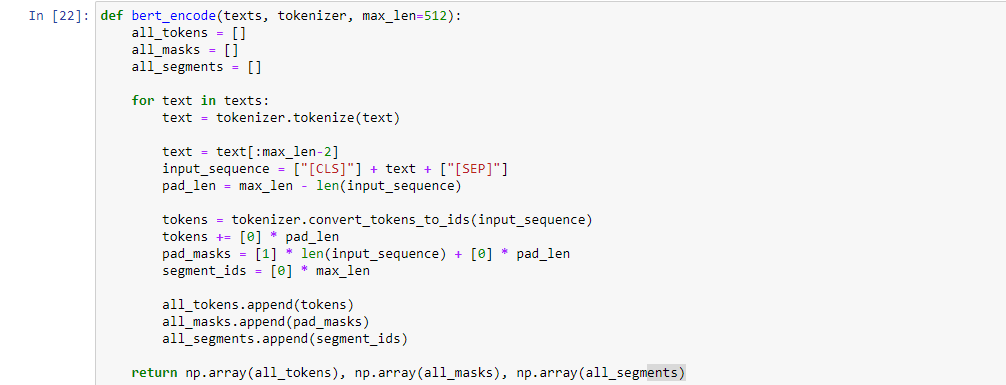
1. **BERT embedding, Training and Fitting:**

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.

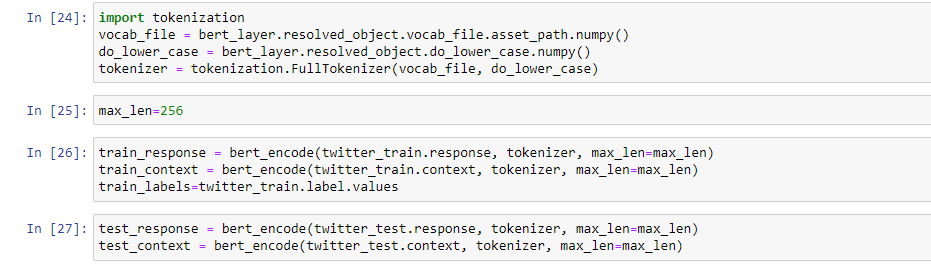
This is where the initially installed tensorflow\_hub comes into play. This model has been pre-trained for English on the Wikipedia and BooksCorpus using the code published on GitHub. Inputs have been "uncased", meaning that the text has been lower-cased before tokenization into word pieces, and any accent markers have been stripped.



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



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.



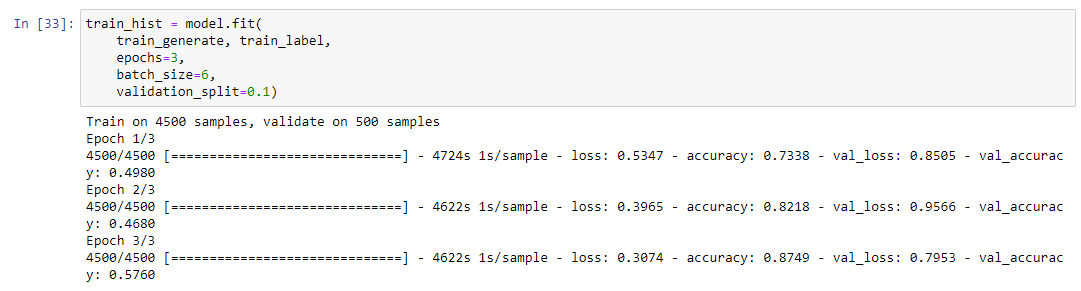
The following code creates a neural network model with the context and the response as inputs. However, we encoded the response and the context separately, and created separate neural network models and later concatenated them together right before they went to the output layer. Lists test\_generate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

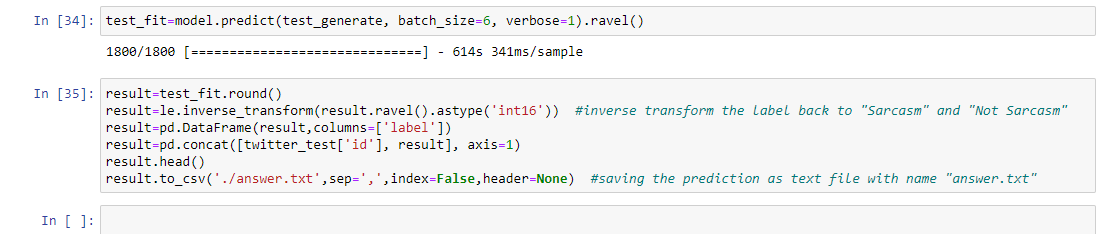
1. **Making prediction based on the trained model**

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.



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.



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.

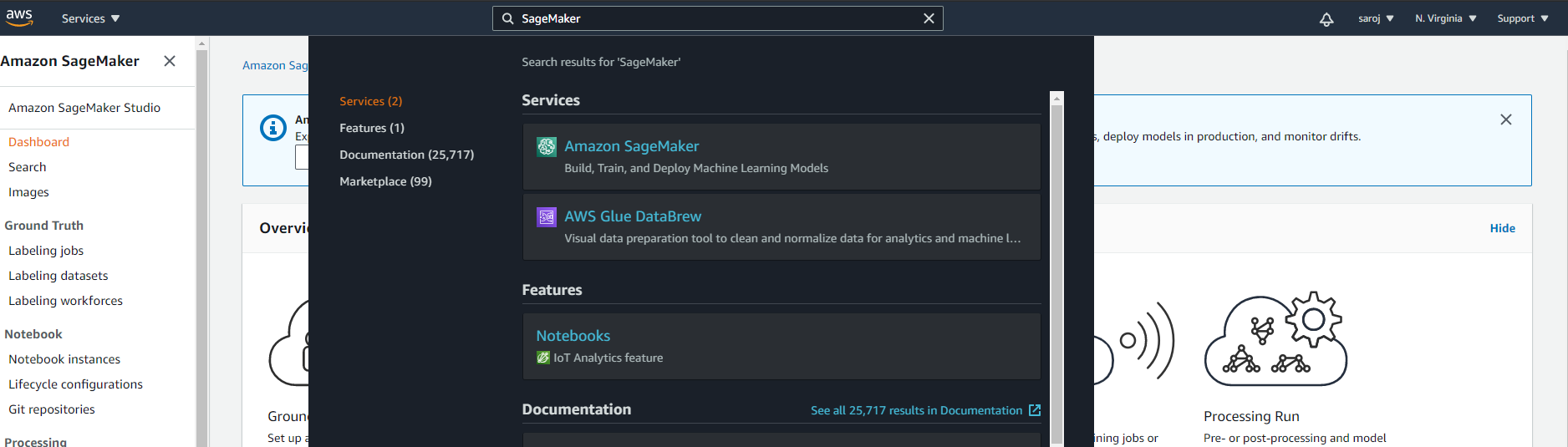


1. **How to run the Source Code?**

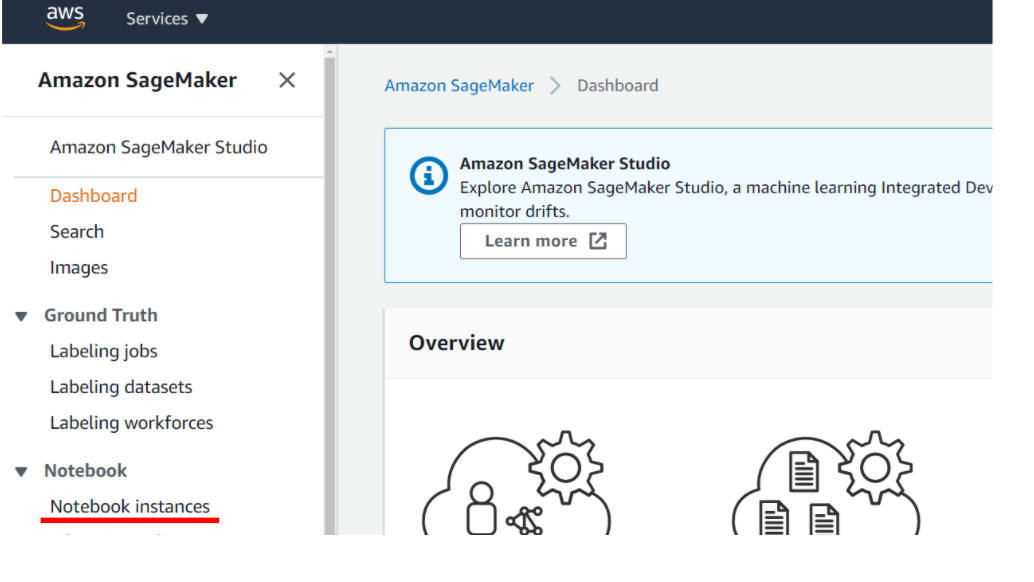
The code will install all the necessary libraries, import all required libraries, therefore there is no need for any additional installation to run this code provided that this code is ran under specific kernel of **“conda\_amazonei\_tensorflow2\_p36”** which is available in Notebook instance of AWS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

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

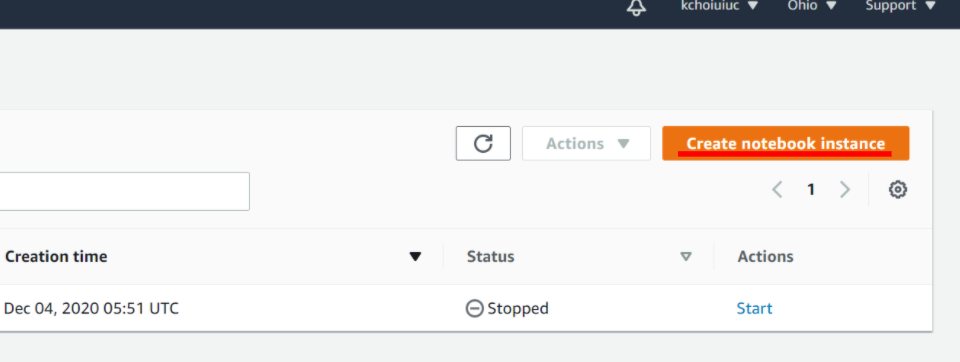
1. Go to <https://aws.amazon.com/> and create an account if you do not have one.
2. Once you are logged in, type SageMaker on the search tab.



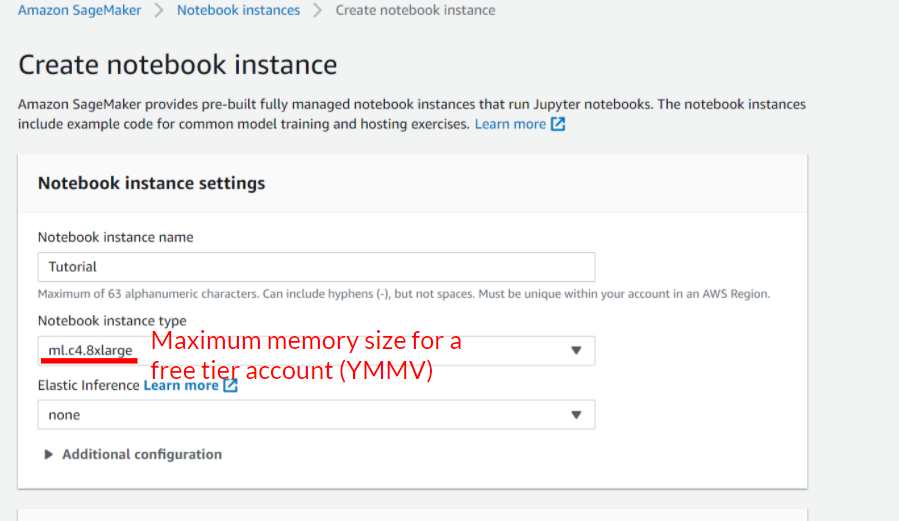
1. Go on Notebook Instances as shown below.



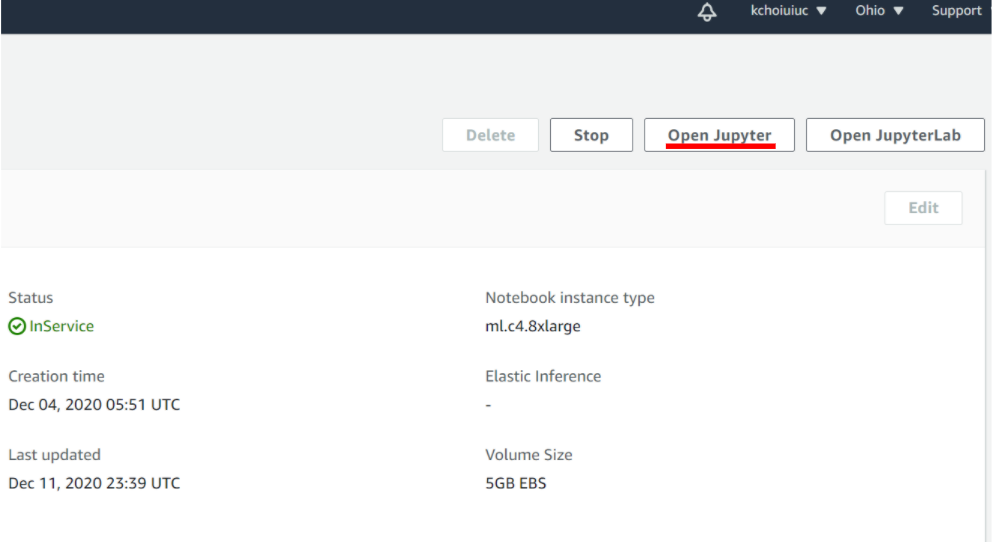
1. Click on Create Notebook Instance



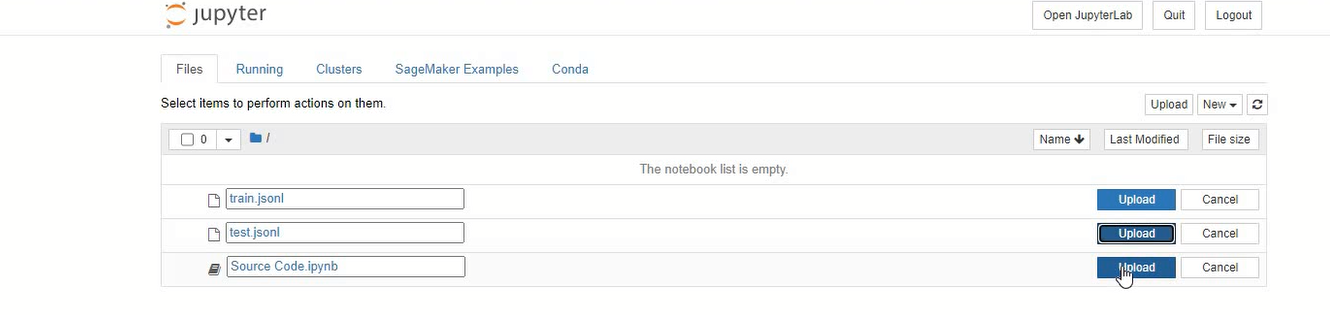
1. Type any name for Notebook Instance name and select Instance type. The free tier AWS version only allows certain maximum size. We use ml.c4.8xlarge to train our model which is available for a free tier account.



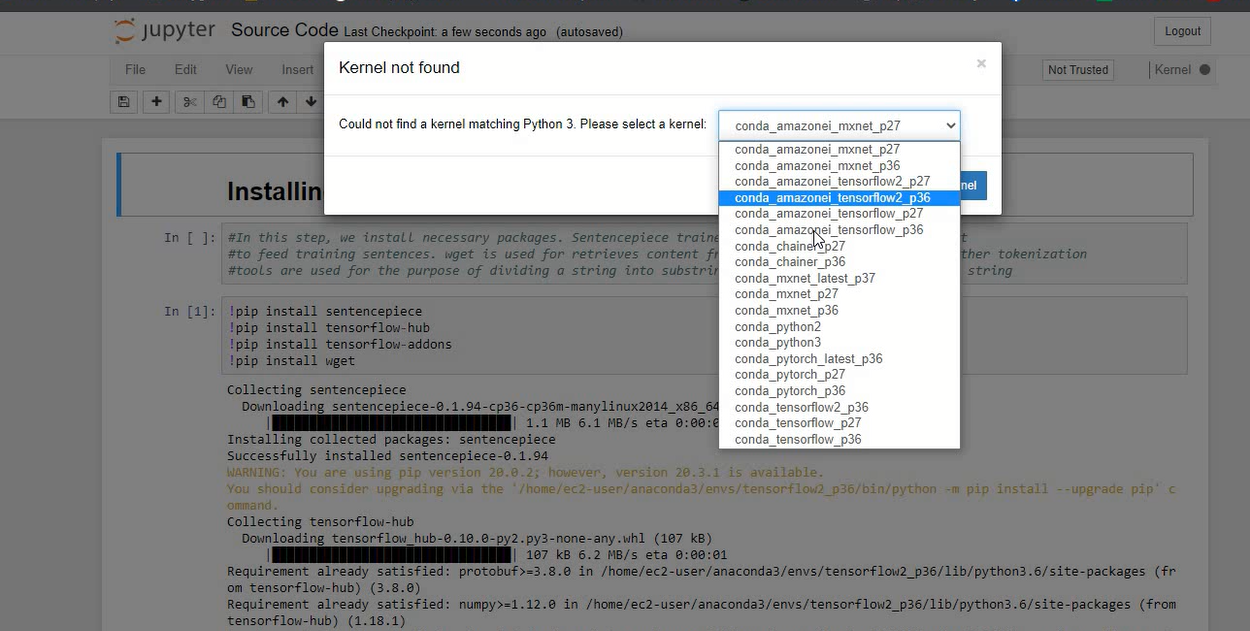
1. Once you create a notebook instance, it will take 1-2 minutes to be activated (inService). Once you see an inService sign, you can click “Open Jupyter” as below. It will open a notebook on your default browser.



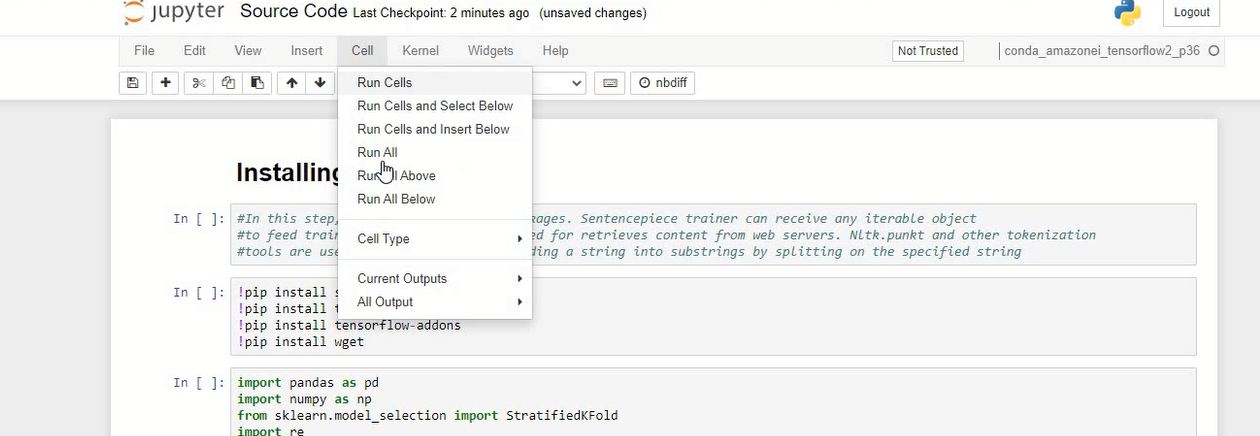
1. Once your AWS notebook opens, upload the source code (notebook file), train.jsonl, and test.josnl file on the notebook.



1. Once your upload is finished, click on notebook (“Source code”).
2. 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.



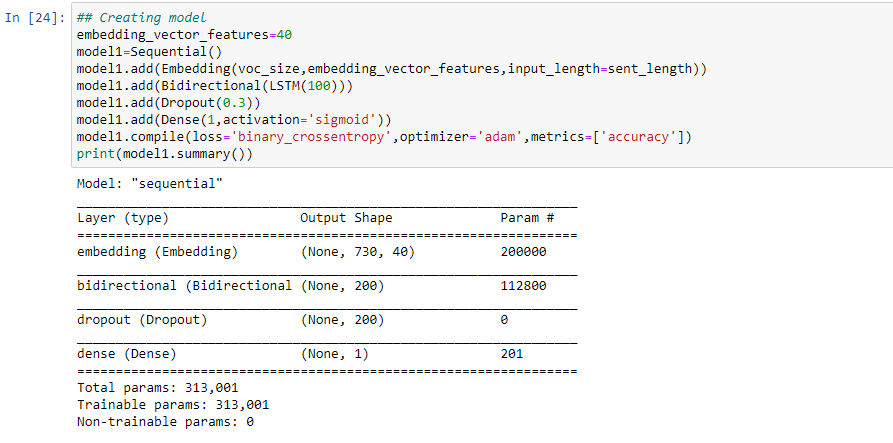
1. Once the notebook is open, go to “Cell” and click “Run All”.



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.



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.

1. Epoch
2. Learning batch size
3. Learning rate
4. Max Length to feed into Bert Layer (max\_length)
5. 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:

1. 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.
2. 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://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://www.kaggle.com/funxexcel/keras-bert-using-tfhub-trial