Graphical user interface, website

Description automatically generated

# Final Project: NLP – Twitter Hate Speech Detection Using Transformers (Deep Learning)

Student Name: Manoj Kumar Thangaraj

E-mail: manojthangaraj92@gmail.com

Module Title: Data & Web Mining

Module Code: LISUM01

Specialization: Natural Language Processing

Contents

[Final Project: NLP – Twitter Hate Speech Detection Using Transformers (Deep Learning) 1](#_Toc80295424)

[1. Project overview/scope 3](#_Toc80295425)

[2. Data Source 3](#_Toc80295426)

[3. Preparing and Pre-processing data 3](#_Toc80295427)

[4. Data Visualization 5](#_Toc80295428)

[5. Transforming the data 7](#_Toc80295429)

[6. Model Building 9](#_Toc80295430)

[7. Training and Validation 11](#_Toc80295431)

[8. Evaluation and Metrics 12](#_Toc80295432)

[9. Model Inference 13](#_Toc80295433)

[10. Conclusion 15](#_Toc80295434)

[11. Reflections on learning 15](#_Toc80295435)

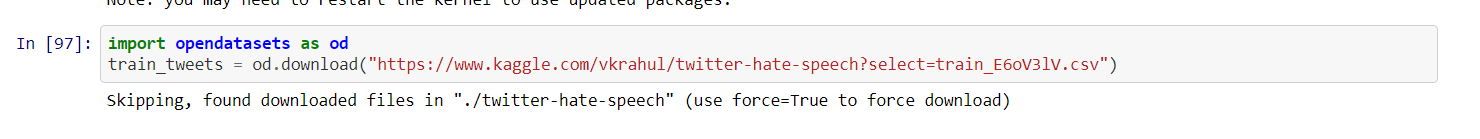
[12. References 15](#_Toc80295436)

# Project overview/scope

The main aim of this project is to develop a hate speech detection model using Transformers architecture in deep learning. The term hate speech is understood as any type of verbal, written or behavioral communication that attacks or uses derogatory or discriminatory language against a person, or group based on what they are, in other words, based on their religion, ethnicity, nationality, race, colour, ancestry, sex, or any other identity factor.

It is basically a form of Classification task. Therefore, for training, a model that can classify hate speech from a certain piece of text can be achieved by training it on a data that is generally used to classify sentiments.

# Data Source

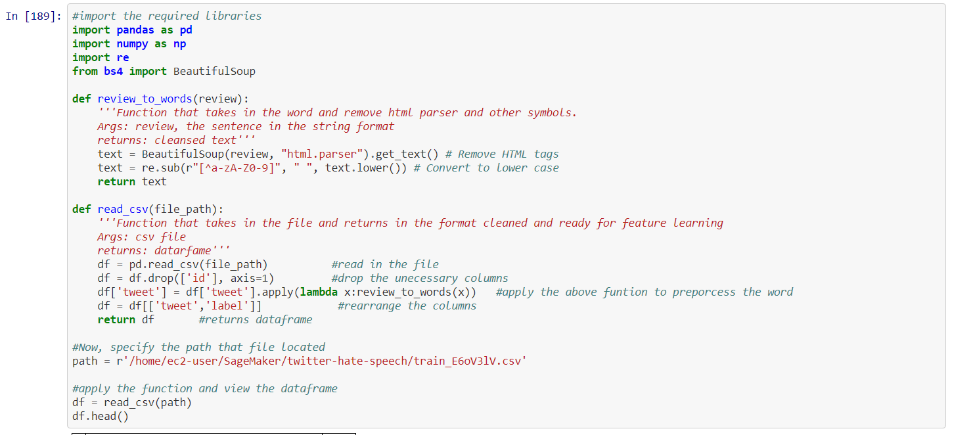
The data source is given by the company. The dataset is available in the following <https://www.kaggle.com/vkrahul/twitter-hate-speech?select=train_E6oV3lV.csv>. We express out gratitude to Mr. Rahul Agarwal for this dataset. The dataset is downloaded on the Jupyter Notebook using the open-source library OpenDatasets.

# Preparing and Pre-processing data

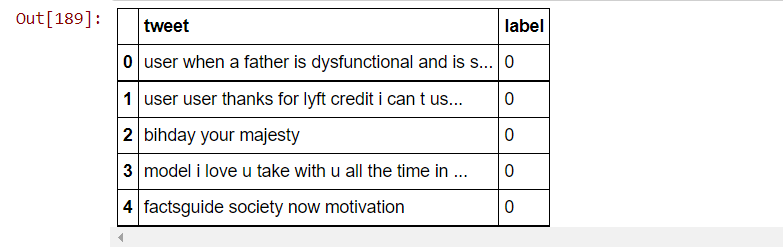
The important step of this whole project is pre-processing the data. This is very important in terms of replacing the words, symbols, HTML tags that are not meaningful in anyway. These items present in the data will lead to false predictions, directing the training in a wrong way.

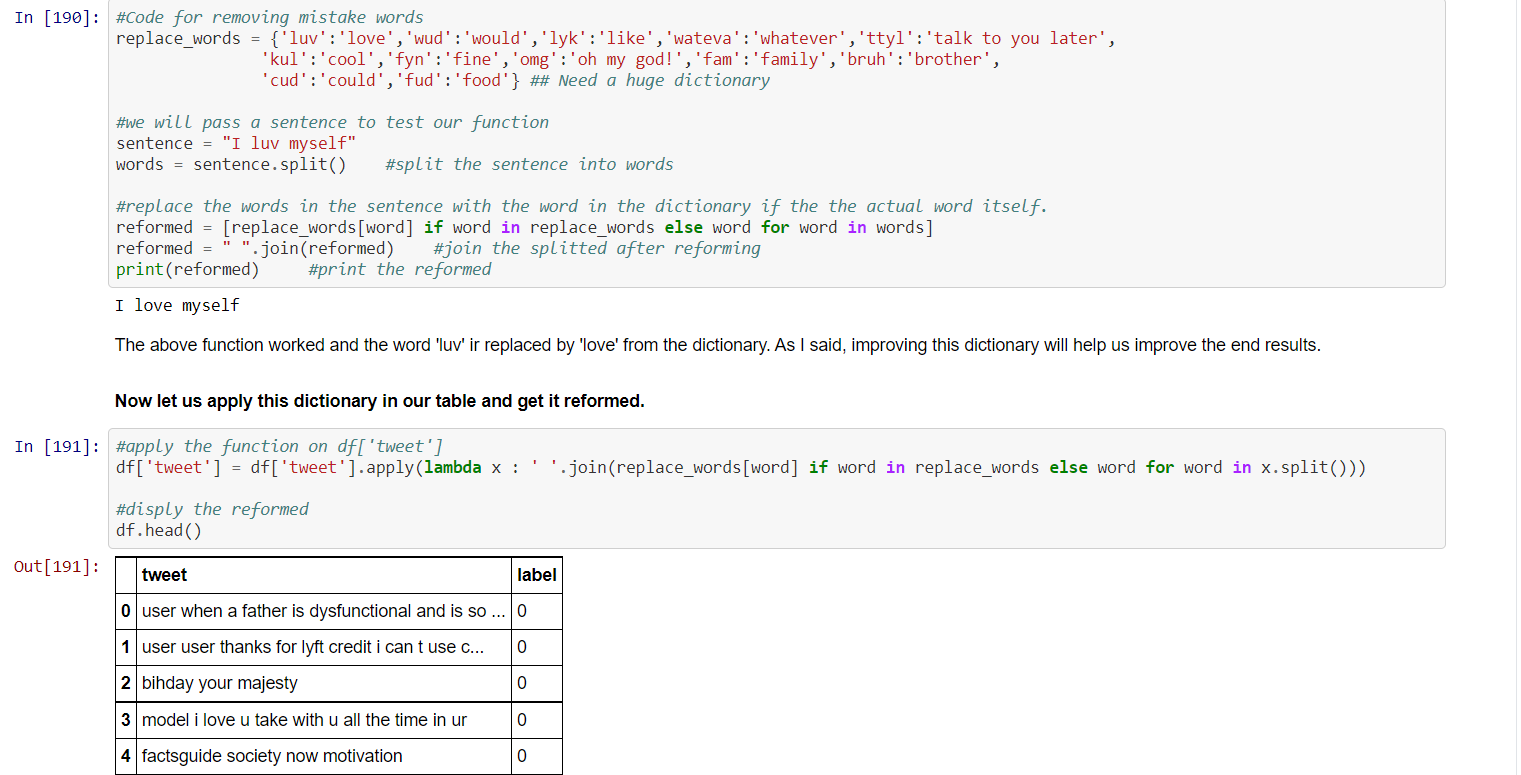
The library BeautifulSoup and Regex has been used to take out the words that are not meaningful. The function review\_to\_words is defined to do this job. This function will return the processed text back to us.

In addition to that, function read\_csv is defined to take the above function and do some additional steps before we can get the full data. The snippet of the code is presented down below.



The snippet of the output is given below.

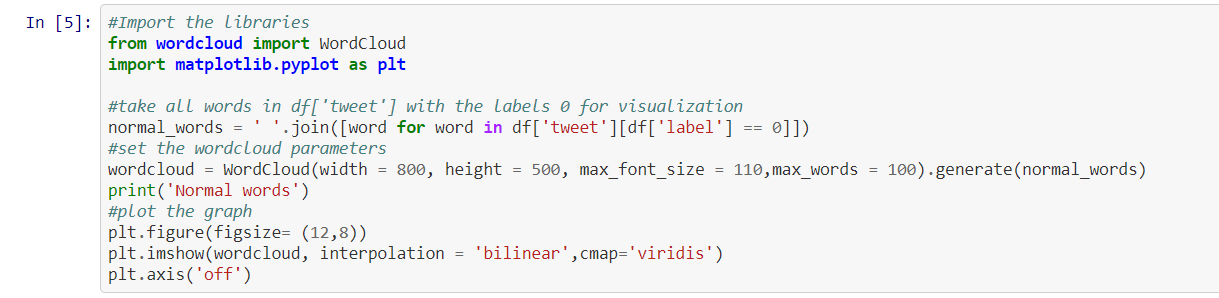


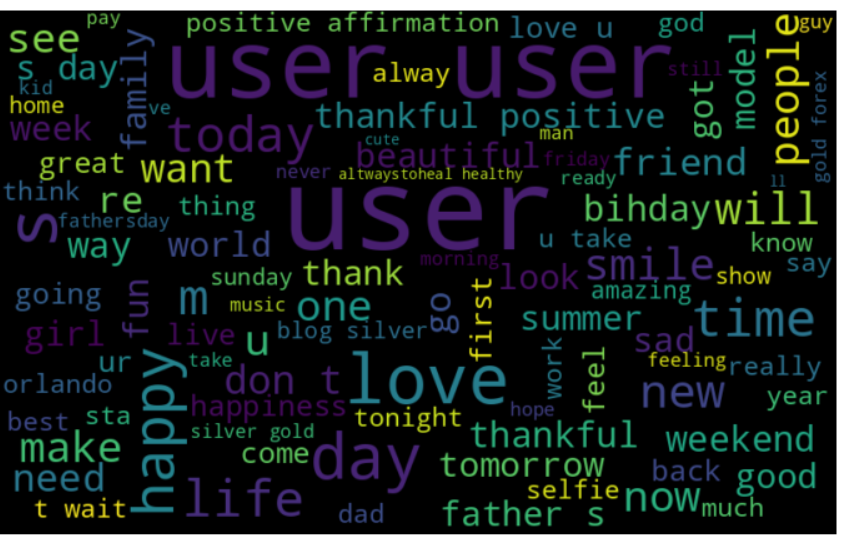
Once the above process is finished and we got our results, a new dictionary has been developed for locally mistyped words and their correct words as replacements. Lambda function is developed to send these keys, value pairs to look for keys that are present in the dataset and to replace them with their values in the dataset. Developing this dictionary will eventually improve the results.

# Data Visualization

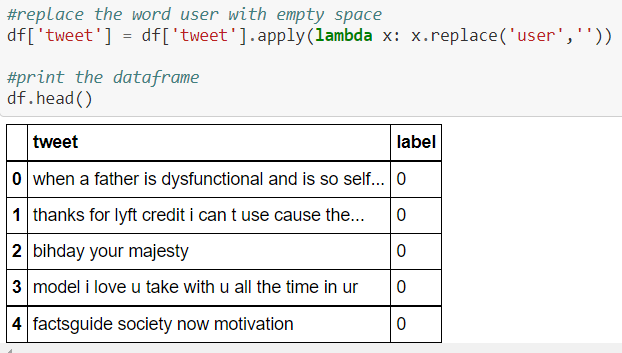
Though it’s the textual data, there is always way to visualize data. In this part we picked up the wordcloud library to show the usage of the words. Eventually, it helped us to go back to the processing part and do some further processing to the good quality data.

Also, the balance level between the labels has been checked and found there is high level of imbalance found between the labels. This is something to be addressed.





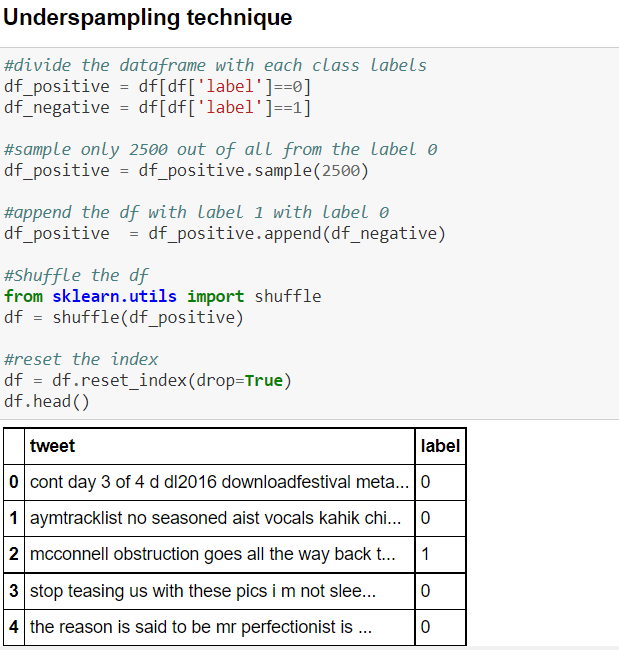
Upon visualizing the picture, we found that the word ‘user’ has been used many times. This is adding no meaning to text and needs replacement. It can be added to dictionary with empty space as value which is not an ideal way to do it. Therefore, we defined a lambda function to replace this word with empty space.



There is high Imbalance between the label 0 and 1. To tackle this problem, we have many ways to do it. There is two popular ways to this is \*oversampling\*\* by increase the number tweets with label 1. We can do this with the help of SMOTE overspampling technique. But if we can see, it's approximately 12 times higher than label 1. In our case, upsampling might lead to overfitting in the training.

The another is undersampling, which is good for this type of problem. The disadvantage in picking them up is we are going to loss so much information in training this model. But upon development, and collection of data in the future and updating the model, this problem can be resolved.

Therefore, we are picking undersampling technique for this problem and continue our model development and training process.



# Transforming the data

Deep learning models generally prefer their training data to be represented by numbers i.e., tensors in arrays or NumPy arrays. For this requirement, the text formats in our dataset needs to be transformed into tensors which our deep learning model can communicate to. In other words, tokenizing each word present in our dataset based their number occurrences in the corpus. Also, fixing the sequence length so that the model knows all the inputs comes in that range. Padding for the minimum occurrences and truncating the largest ones will help us do this.

Padding is a process of adding an extra token called padding token at the beginning or end of the sentence. As the number of the words in each sentence varies, we convert the variable length input sentences into sentences with the same length by adding padding tokens. Padding is required since most of the frameworks support static networks, i.e., the architecture remains the same throughout the model training.

Torch is the important package which is used tot transform to tensors and mathematical operations on it. TorchText is a Natural Language Processing (NLP) library in PyTorch. This library contains the scripts for pre-processing text and source of few popular NLP datasets.

We will define a function build\_vocab, which has the following operations. Before that, there are two different types of fields, Objects-Field and Label-field. The former is the field object from the data module is used to specify pre-processing steps for each column in the dataset and the latter is the special case of field object which is only used for the classification tasks.

When we instantiate these fields first, there are parameters needs to be considered,

* Tokenize: specifies the way of tokenizing the sentence i.e., converting sentence to words. I am using spacy tokenizer since it uses novel tokenization algorithm
* Lower: converts text to lowercase
* batch\_first: The first dimension of input and output is always batch size

Following the above steps, we will create a list of tuples in which every tuple contains object field and followed label field as defined above. In fact, we will arrange the tuples in accordance with our columns of the csv file. Since we have index as the first column, we will specify the tuple (None,None) to ignore that column.

Once we are done with instantiating the fields, we will load the pre-processed dataset in the torchtext tabular dataset function with the specified parameters. Now it’s the time to split the dataset into train and validation dataset. We don’t need test dataset since we have a separate test dataset. This is done with the help of random library with its split function which is excellent for splitting up the dataset.

The next step is to build the vocabulary for the text and convert them into integer sequences. Vocabulary contains the unique words in the entire text. Each unique word is assigned an index. Below are the parameters listed for the same

Parameters:

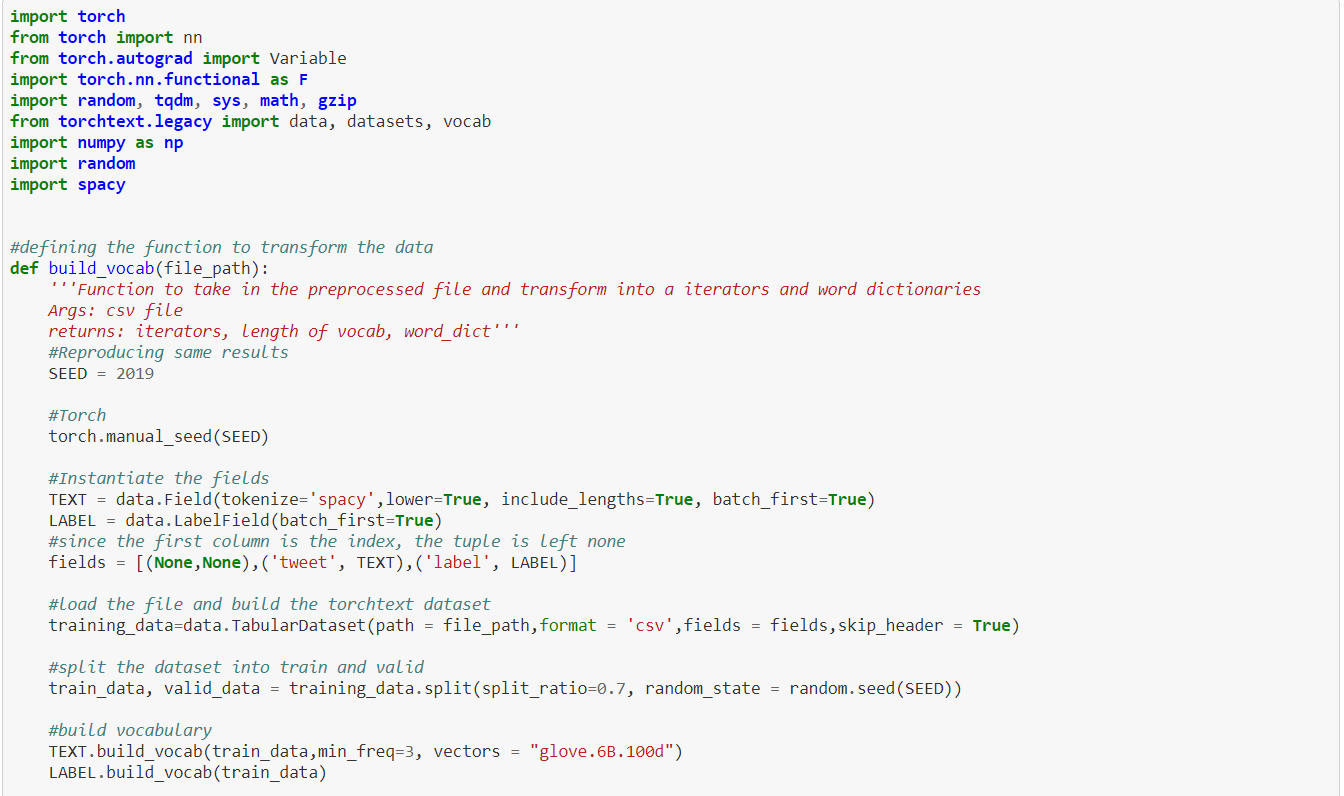
1. min\_freq: Ignores the words in vocabulary which has frequency less than specified one and map it to unknown token.

2. Two special tokens known as unknown, and padding will be added to the vocabulary

* Unknown token is used to handle Out of Vocabulary words
* Padding token is used to make input sequences of same length

Let us build vocabulary and initialize the words with the pretrained embeddings. Ignore the vectors parameter if you wish to randomly initialize embeddings.

Now we will prepare batches for training the model. BucketIterator forms the batches in such a way that a minimum amount of padding is required. Once, the above process is completed, we will get returns train and test iterator objects, word dictionary and length of the vocabulary. The whole function snippet is given below.



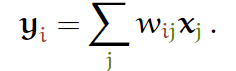


# Model Building

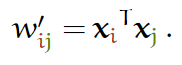
For this project, we aim to build the transformers from scratch. i.e., without using any pre-trained models such BERT etc., Transformer are the very exciting family of machine learning architectures.

Transformers model is built with the help of PyTorch neural networks. For our model, we will build a self-attention layer which is embedded in transformer block which is then embedded in Classification Transformer Model. This is the high-level view of our transformer model.

We will go by explaining one by one. Firstly, the fundamental operation of any transformer architecture is the self-attention operation. It is a sequence-to-sequence operation where a sequence of vectors goes in, and a sequence of vectors comes out. Let’s say vectors X1, X2, X3…. Xt, have their output vectors as Y1, Y2, Y3….Yt. The self-attention operation takes a weighted average over all the input vectors.



Where j indexes over the whole sequence and the weights sum to one over all j. The weight wij is not a parameter, as in a normal neural net, but it is *derived* from a function over 𝐱i and 𝐱j. The simplest option for this function is the dot product



The dot product gives us a value anywhere between negative and positive infinity, so we apply a SoftMax to map the values to [0,1] and to ensure that they sum to 1 over the whole sequence. This is the basic operation of self-attention.

But the dot product is not possible when there are not any features. In our case we have only text data. For this problem, we have a concept called key, value, query concept where the query is the input vector key and values are vectors that is matched against input vectors.

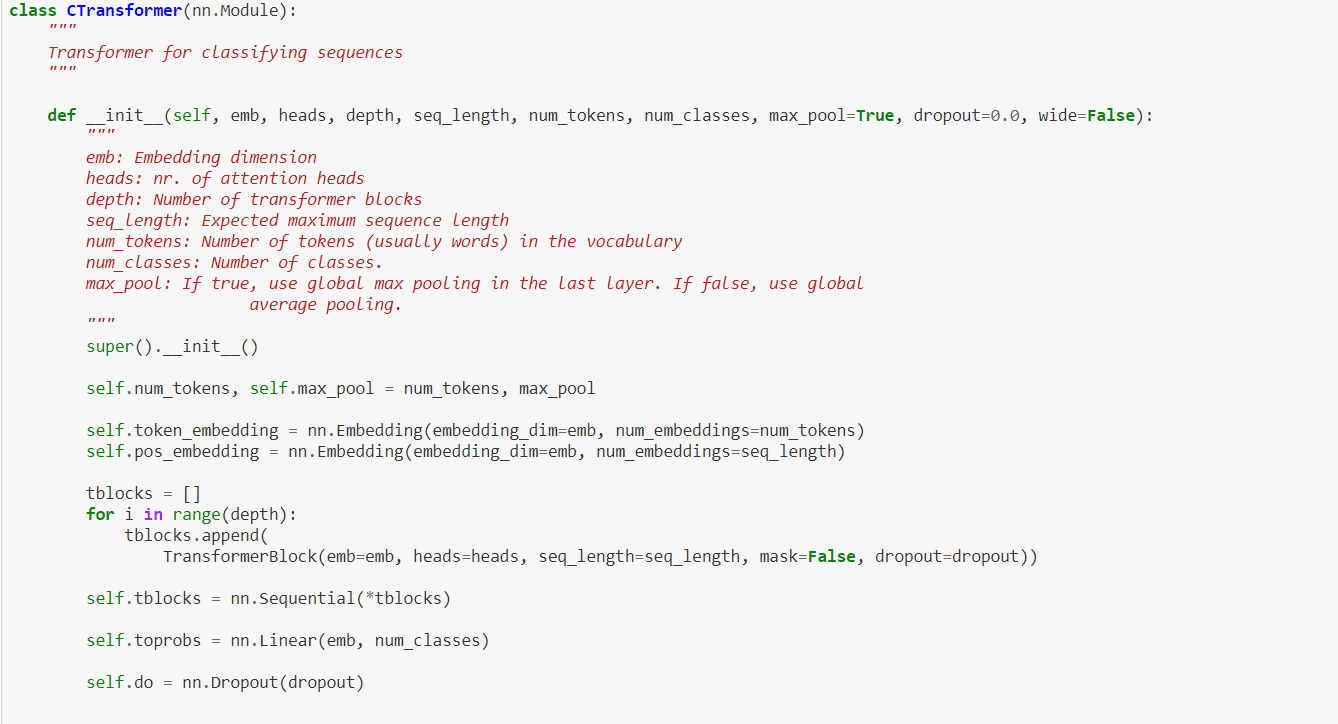
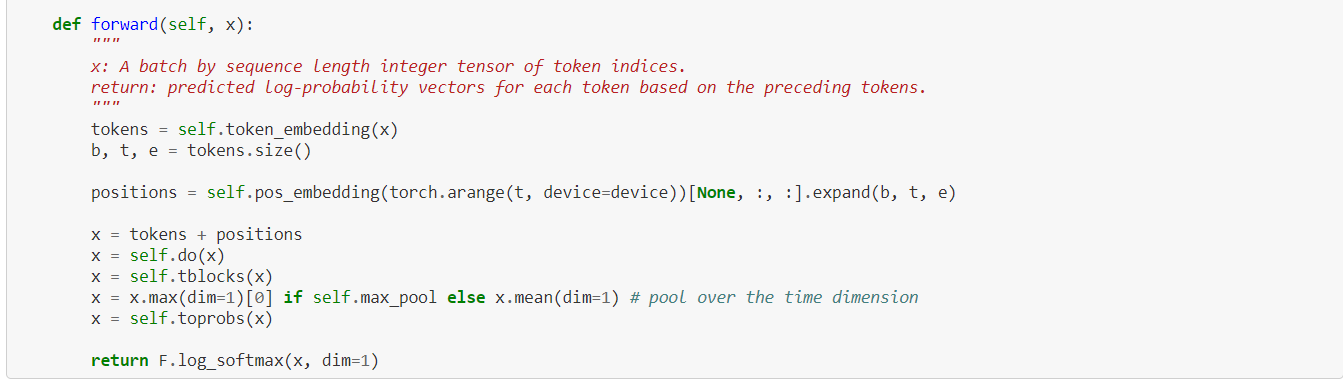
We will create this simple transformer as we go along. First the self-attention in pytorch. The first thing we should do is work out how to express the self attention in matrix multiplications. A naive implementation that loops over all vectors to compute the weights and outputs would be much too slow.

We’ll represent the input, a sequence of t vectors of dimension k as a t by k matrix X. Including a minibatch dimension b, gives us an input tensor of size (b,t,k).

The set of all raw dot products w′ij forms a matrix, which we can compute simply by multiplying X by its transpose. Then to turn raw weights into the positive values that sum to one, we apply a row-wise SoftMax. Finally, to compute the output sequence, we just multiply the weight matrix by X. This results in a batch of output matrices Y of size (b, t,k) whose rows weights are sum over the rows of X. This two-matrix multiplication and one SoftMax gives us a basic self-attention.

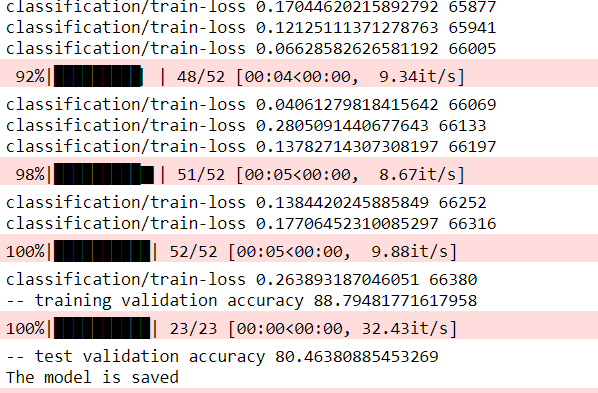
Now we will build a transformer block, this transformer block will have a self-attention layer, a feed forward layer and another layer normalization. Residual connections are added around both, before the normalization. The important thing is to combine self-attention with a local feedforward, and to add normalization and residual connections.

Finally, now we will build the classification transformer. We will build a transformer blocks according to the number of depths. Before that whatever the tensors coming in, will have their own tensor value, along with their positions. For this we will have a positional embedding layer which will give a position to each word which is then added the embedding tensors. This will go through the dropout layer and the number of transformer blocks and finally to SoftMax value.



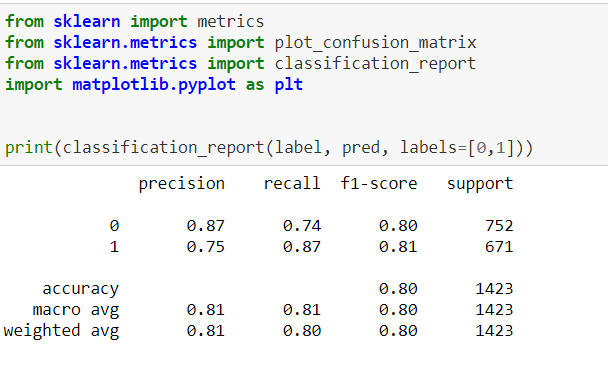
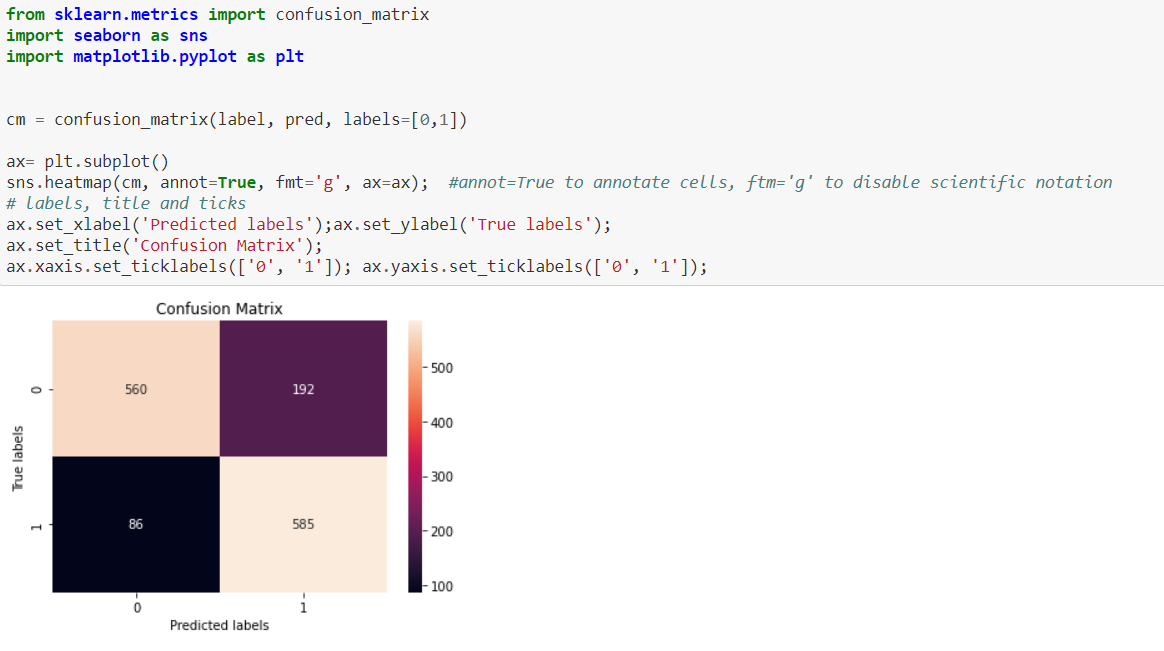
# Training and Validation

Training the model and validating the model is done using same loop. The classification loss and training validation is calculated during training and the test validation accuracy is calculated during the validation. Also, the original labels, predicted labels obtained to do evaluation and metrics.

The training and validation yielded a good result. For the 20 epochs, the training validation came up to 88.79% and test validation came up to 80.4%. The model is then saved for model inference.

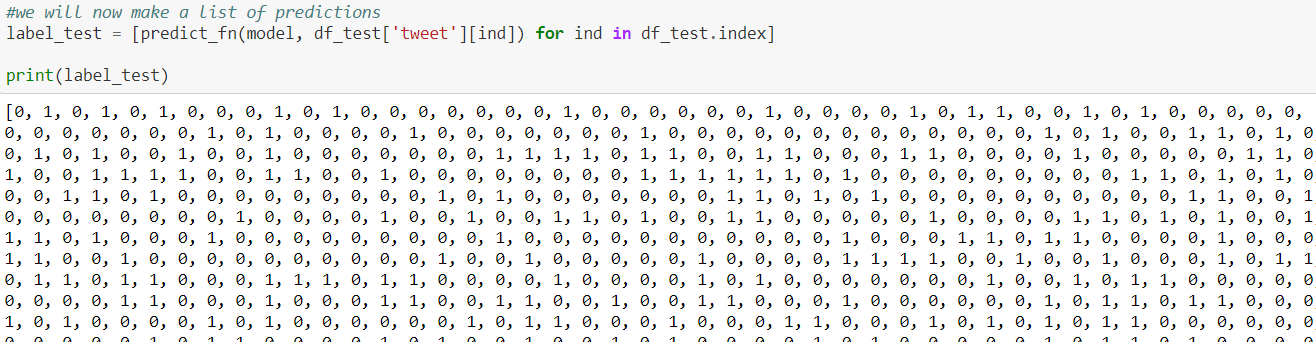
# Evaluation and Metrics

From the training and validation process, the original labels and metrics are extracted. They are saved in a NumPy arrays. The general evaluation metrics such as confusion matrix, precision, recall and f1 scores are compared.

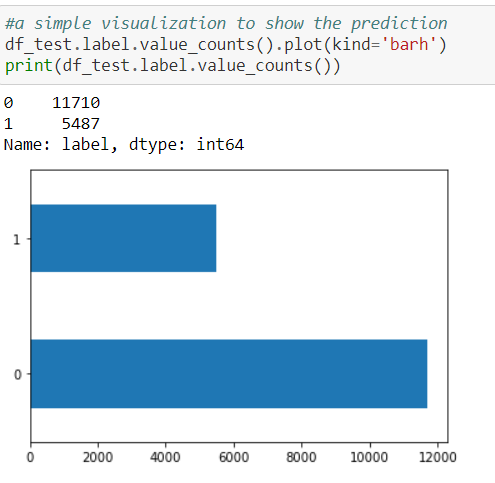
In terms of precision, which is true positives over the true positive and negatives, the label 0 achieved 87% and label 1 achieved 75%. On the other hand, recall which is true positives over the predicted positives, label 0 yielded 74% and label 1 achieved 87%. Therefore, the f1 is 80% for both. The confusion matrix is also shown below.

# Model Inference

Once all the training and validation is done, the trained model is tested against the unseen data. First through a single sentence to see how it behaves and through the test dataset. The label 1 sentence has been predicted correctly.

The dataset was loaded, pre-processing done and sent through the prediction function. The resulting output is appended to the test dataset and written into a csv file.

The count plot shows us the number of 0 and 1 labels predicted in the test dataset along with the values.



# Conclusion

The twitter dataset had a raw text with so many noises. It has been processed, imbalance has been dealt and sent it through the feature extraction. Model was designed and training and validation was done. This model was able to predict 80% of the original tweets, that it had a hateful or non-hateful contents. It predicted 20% wrongly. It is a cyclic process where I must go back to text processing to do stemming and lemmatization, replacing the slang words with original English words, do a different balancing method and fine tuning the model would give me better results or might behave worse. But until I get much more efficient result, I must keep working on improving the model which is the goal.

# Reflections on learning

Through this project, I had a wonderful opportunity to learn about Natural language processing, different types of text processing, feature extraction techniques. It added a great experience in my journey of Machine Learning and AI.

# References

* A blog by Peter Bloem on Transformers, [link](http://peterbloem.nl/blog/transformers).
* Text Classification using PyTorch by Analyticsvidhya, [link.](https://www.analyticsvidhya.com/blog/2020/01/first-text-classification-in-pytorch/)
* PyTorch Documentation, [link.](https://pytorch.org/docs/stable/index.html)
* Attention is All you need, research paper by a google team, [link](https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf)
* Smote for Imbalanced classification method by Towards Data science, [link.](https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/)
* How to validate a model, Towards data science, [link.](https://towardsdatascience.com/why-and-how-to-cross-validate-a-model-d6424b45261f)