# Tweet Sentiment Analysis

(Summer-2018)

**~Divyam Srivastava~**

I recently worked on a project that uses Deep Learning concepts.

**1) Problem Statement:** To perform **Sentiment Analysis** on Tweets.

**2) Dataset Details:** Some random twitter data was downloaded from Kaggle website which consisted of standard **text** files separately for both positive and negative tweets each **1250K** in number. Each tweet in the text file was separated with a newline character.

**3) DeJunkifying Data:** Firstly I accessed all the tweet sequences from both the files and appended them together to have a 2-D word array of total rows equal to **2500K(1250K*2)** containing both positive and negative tweets. Next I removed "stop-words" (which I came to know they hardly make any difference in sentiment analysis) and removed any left out punctuations/special characters as a part of pre-processing (/cleaning) the data from that array. After that I tokenized all of words in the array using **Keras Tokenizer,** shuffled all the data followed by splitting of data into training and testing sets (**80:20 ratio)** and post-padding all the required tweet sequences(as all tweets are definitely of not same size) to form a proper matrix to feed into the network.

**Note:** Above data was created for word level analysis of tweets i.e., Tweets were stored in the matrix row wise separated on word level. A similar matrix was created for character level analysis of tweets by storing the tweets in the row wise manner but separated on character level.

Finally I was left with the following:

**Word level Data: Training input matrix – 2000K X 67 Testing Input Matrix – 500K X 67**

**Training output matrix – 2000K X 1 Testing Output Matrix – 500K X 1**

**----------Output Matrix contains either "1" for positive sentiment or "0" for negative sentiment.**

**Character Level Data: Training input matrix – 2000K X 62 Testing Input Matrix – 500K X 62**

**Training output matrix – 2000K X 1 Testing Output Matrix – 500K X 1**

**4) Model Selection:** Now everything boils down to model development. I had already used CNN models to classify Images, but some articles I came across highlighted its use in **NLP** so I decided to test it right

away. As each tweet is represented by a single row in a matrix (series of random tokens), so to train the model on the tweets I decided to use **One-Dimensional Convolutional filter and One-Dimensional Max-Pooling layer. Window size of each filter/layer was tested and finalized either through trial or as mentioned in articles/research papers.**

**Note: I had used [Optimizer: Adam, Test Data: epochs=5, batch size=1000] for all my subsequent models. Keras API is used for building the models.**

**Model 1: Normal Feed forward architecture. (Word Level)**

I started with a standard model of stacked up **Convolutional+Maxpool layers** on the **test matrix.** These convolutional filters help in extracting out the most important features automatically by adjusting filter weights after each loss optimizations update. Max pool layers further reduces the number of these features by taking out a max value from certain group of features whose size depends on the desired window level. These final set of features are flattened to a single dimension before feeding them to the neural network.

**Conv-1D → MaxPool-1D → Conv-1D → MaxPool-1D →Flatten all features to a single dimension →Dense layer (Group of ReLU activated neurons)→ Dropout Layer (to avoid over-fitting) → single neuron (sigmoid activated)**

**All Convolutional layers are ReLU activated** to introduce non linearity in the model. Research shows that any curve can be broken down into a number of ReLU curves, hence more and more ReLU layers can take our **predicting curve** more towards **the training curve** but one must be careful not to over-fit the data in such a way.

The testing data resulted in an accuracy of around **60%** which is **fairly very poor**.

**Conclusion**: There's a huge loss of semantic information in this model. This made me come across the term "**Word Embeddings"**.

Till now each tweet in my matrix was represented as a series of random integral tokens occupying space only in a single dimension. This can lead to a bad classification accuracy as there will be **no** concrete decision lines trying to classify sentences all lying in the same dimension.

A deeper dig into the topic made me realize I can actually enhance the **Spatial** representation of my words (or sentences) by converting all the word tokens into a definite length **N-dimensional** vector. In this way each sentences will now be represented by high dimensional word vectors which will surely preserve the semantic information in a much more efficient way and hence more complex decision boundary can be easily modeled out for better classification.

**Model 2: Feed Forward architecture + Word Embedding layer (Word level)**

Here I used an **almost same feed forward network** as was used in the previous model with some minor Hyper-parameter tuning after referring to two research papers.

**Word Embeddings →Conv-1D →Conv1D → MaxPool-1D → Conv-1D → MaxPool-1D →Flatten all features to a single dimension →Dense layer (Group of ReLU activated neurons)→ Dropout Layer (to avoid over-fitting) → single neuron (sigmoid activated)**

Now I **didn't** chose bag of words model or a pre-trained word embeddings as they were certainly **not** accustomed on my own data. I instead used an **Embedding layer from the Keras API** with a preset dictionary level and a vector length of my desire (which again was tuned many times during testing the model for avoiding over-fitting).

**Note**: Suppose my each tweet is having dimension **1 X N**, where **N** represents the number of word tokens. Now if I wish to apply word embedding layer on this tweet with a **vector size** (No of higher dimensions) of my desire say **D,** then each tweet data will transform into a **D X N matrix** which can be interpreted as adding D-dimensions to each word tokens. My sentence now occupies a unique space on a **D-Dimensional** plane (contrary to occupying only a single dimension earlier).

The test data was now giving a fairly good accuracy of around **83%**.

**Note:** Although the accuracy was high enough, yet a plot of **training (and validation) accuracy v/s epochs** clearly indicated **over-fitting** which disqualifies this model from final deployment.

**Conclusion:** Embedding layer certainly helped preserve semantic info (on word level), but is not robust for the wild data. A possible reason can be that the test (wild) data has some totally new words (or new form of words existing in training set) which were not there in the training data which indicates the shortcomings of word level sentiment analysis. I then came across another interesting method of "character level" classification which involves breaking the entire tweet sentences into characters (**not words).**

**Model 3: Feed Forward architecture + Character embedding layer ( Character Level)**

Here I used the same feed forward network (again with some fine hyper-parameter tuning based on trial) with a **character level embedding** layer at the top. This time I fed to the network character level dataset (which I prepared from the original dataset).

**Character Embeddings →Conv-1D →Conv1D → MaxPool-1D → Conv-1D → MaxPool-1D →Flatten all features to a single dimension →Dense layer (Group of ReLU activated neurons)→ Dropout Layer (to avoid over-fitting) → single neuron (sigmoid activated)**

This time I was able to get a final accuracy of around **80%** on the test data with **no over-fittings** as such.

**Conclusion:** Characters proved to be a more powerful way to preserve semantic information. Word level analysis limits itself to **particular words or slightly similar words** along with certain word wise dependencies**.** But character level analysis is robust as **character wise dependencies** are very **sensitive** and can easily accommodate wild data.

**This makes my Model 3 qualified to be used as final Production Model.**

**Note:** I am in a process to learn RNNs to incorporate the power of LSTMs and GRUs in my sentiment analysis but until then, **Model 3** will be my final production model.

Here is the link to my well documented Jupyter Notebook for the project:

https://github.com/divyam25/NLP_for_Noobs