Predictive Analytics II Project Report

Real or Not? Disaster Tweets with NLP

Predict which tweets are about real disasters and which ones are not.

submitted in partial fulfillment

of the requirements for the

Course: MSiA 420 (Winter 2020)

Students:

Aditya Tyagi

Bhavya Kaushik

Manish Kumar

Srividya Ganapathi

Table of Contents

I. Introduction

Twitter has become an important communication channel in times of emergency. The ubiquitousness of smartphones enables people to announce an emergency they're observing in real-time. But, it's not always clear whether a person's words are actually announcing a disaster.

Problem Statement: Build a machine learning model using natural language processing to predict which tweets are about real disasters and which ones are not. The objective of this project is to present an effective support for both emergencies and false alarm/rumor countermeasures.

This is an ongoing Kaggle competition with the object of predicting whether a given tweet is about a real disaster or not. If so, predict a 1, If not, predict a 0.This dataset was created by the company figure-eight and originally shared on their 'DataForEveryone'. It is a medium sized dataset with 7613 rows that are divided into train and test sets. The data dictionary is as follows:

Columns

- id-a unique identifier for each tweet
- text-the text of the tweet
- location-the location the tweet was sent from (may be blank)
- keyword-a particular keyword from the tweet (may be blank)
- target - in train.csv only, denotes whether a tweet is about a real disaster (1) or not (0)

In the following data we initially performed some exploratory data analysis and vectorized the tweets using count vectorizer, TF-IDF and Doc2Vec. Then we implemented Logistic regression, SVM, KNN, XGBoost and Neural networks after tuning their hyperparameters through cross validation.

Additionally we also used Recurrent Neural Network for classification of tweets. Used a combination of Embedding layer, LSTM (Long Short Term Memory) layer in the architecture. Since the textual data have a sequence, RNN architecture capture the chronology of the words appearing in the tweets.

We have done all our analysis in Python.

II. Exploratory Data Analysis

The data has 7613 rows and 5 columns.

An example of what is NOT a disaster tweet is -
I wanted to set Chicago ablaze with my preaching... But not my hotel! http://t.co/o9qknbfOFX
And one of a disaster tweet is -
Forest fire near La Ronge Sask. Canada

The class distribution of disaster vs. non-disaster tweets is as follows -

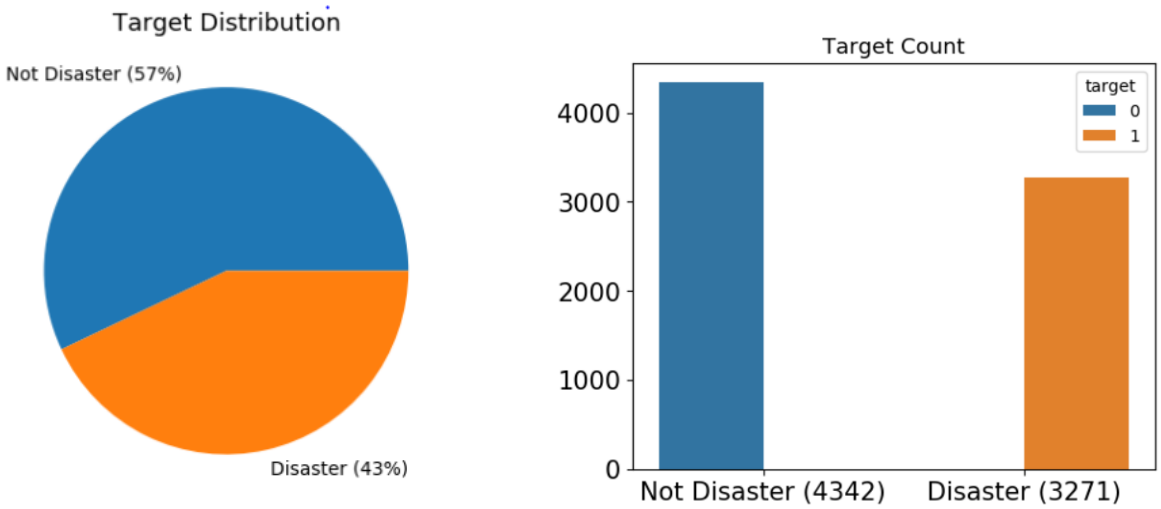

Fig 1: Class balance for the 2 categories

57% of the tweets in the dataset are 'Not Disaster (0)' while 43% are 'Disaster (1)' tweets. The class distribution is fairly balanced.
There was also no missing data in any of the tweet texts, thus, no data imputation was required. We started with a basic analysis at character level,word level and sentence level.

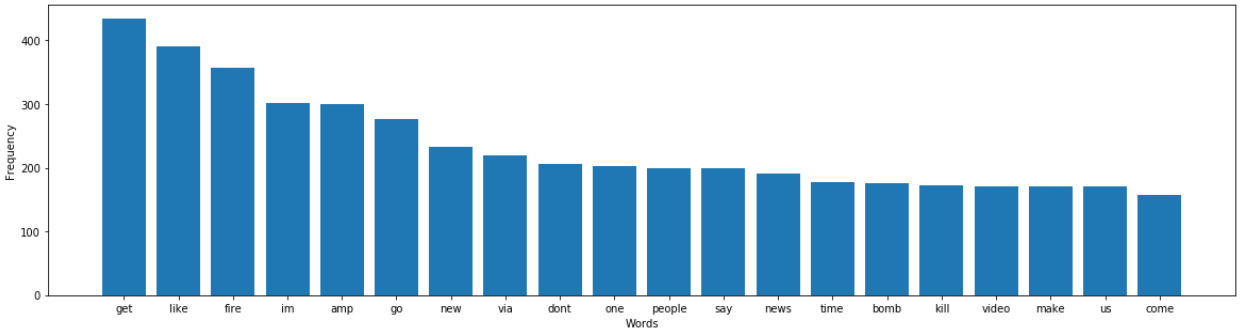

Fig 2: Top 20 most common words

Figure 2 shows the top 20 most common words in the document.

Number of characters in a tweet -

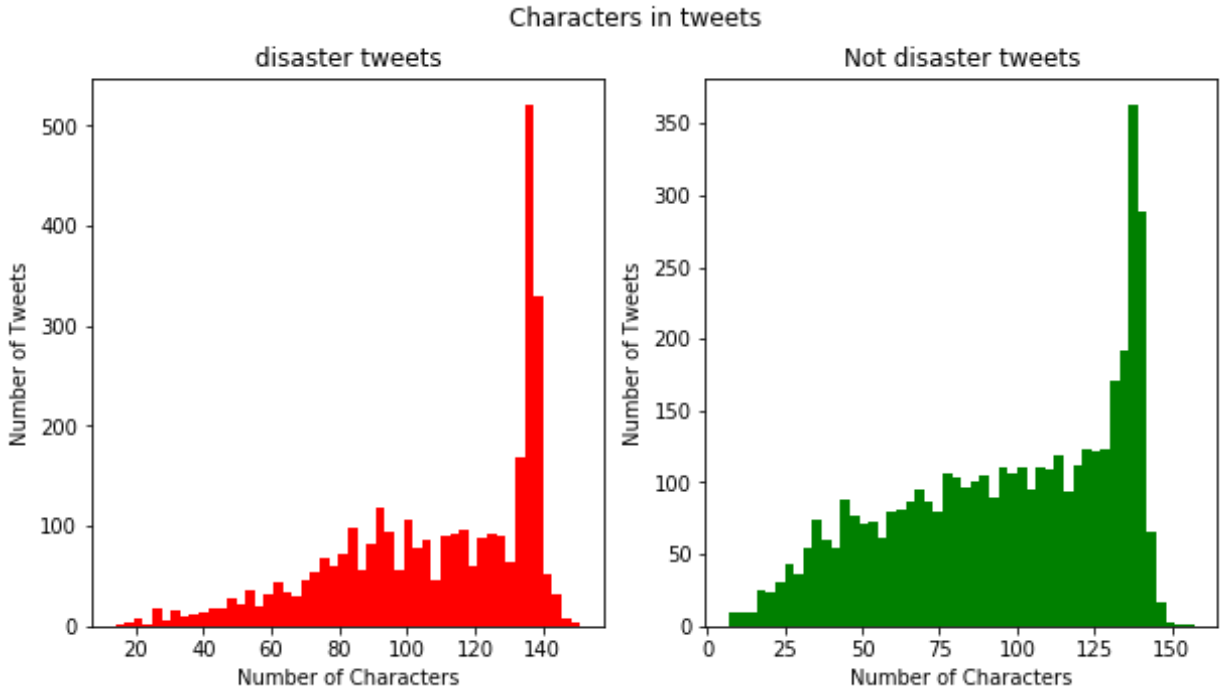

Fig 3: Distribution of number of characters in both the tweets

The distributions seem very similar, both seem to have 125 to 140 characters

Number of words in a tweet

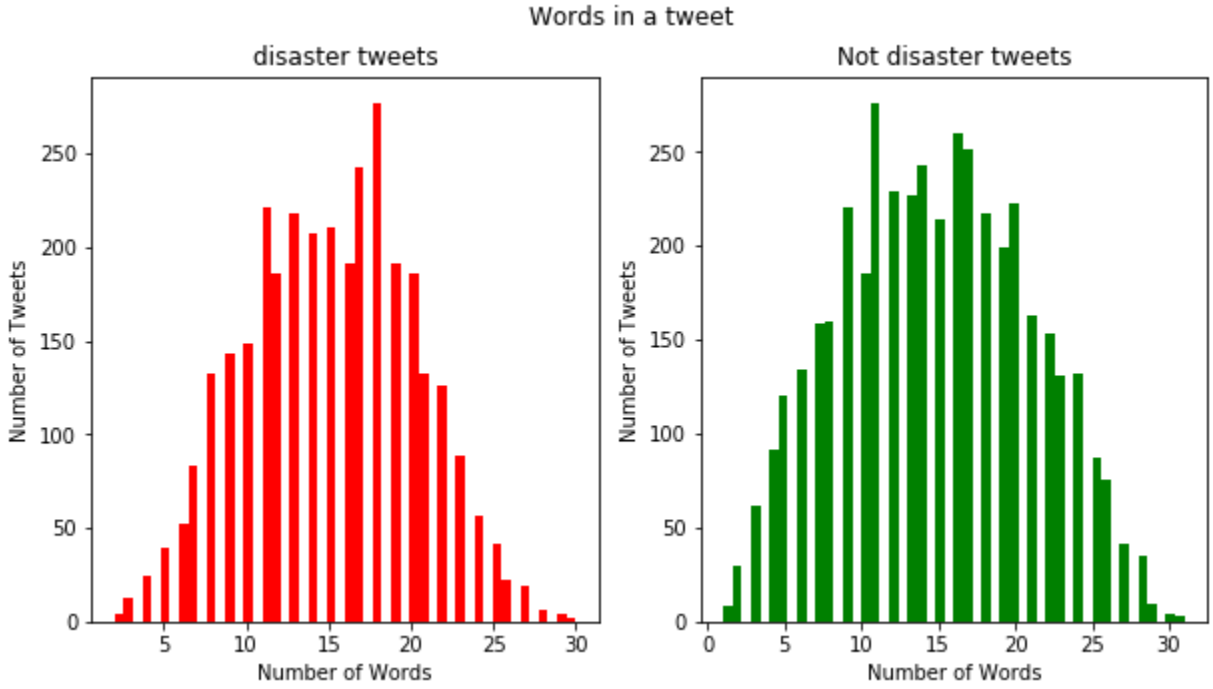

Fig 4:Distribution of number of words in the tweets

Not much can be inferred from the plots above.

We then cleaned the data by removing the following:

- ➔ URLs
- ➔ Punctuations
- ➔ Digits
- ➔ Stopwords (a, and, the, but etc.)

After these steps, all tweets were converted to lowercase.
We performed lemmatization on the cleaned text of the tweets. Lemmatization is the process of grouping together the inflected forms of a word so they can be analysed as a single item, identified by the word's lemma, or dictionary form.

This dataset was now ready for vectorization and we used the count vectorizer, TF-IDF vectorizer and Doc2Vec.

Each tweet was first tokenized. String **tokenization** is a process where a string is broken into several parts. Each part is called a token. For example, if "I am going" is a string, the discrete parts—such as "I", "am", and "going"—are the tokens. A tweet contains only some of those unique tokens - all of the non-zero counts are the tokens that DO exist in the first tweet in a vector with size as the number of unique tokens in the corpus of all tweet texts.

Count vectorizer: It is one of the most simple ways to represent text data numerically using one hot encoding. We create vectors that have a dimensionality equal to the size of our vocabulary, and if the text data features that vocab word, we put a one in that dimension. Every time we encounter that word again, we increase the count, leaving 0s everywhere we don't find the word even once.
In our dataset there are 14922 unique tokens, thus, this is our vocabulary size.
Note - Prior to the data cleaning the vocabulary size was 21367 words.

Term Frequency - Inverse Document Frequency(TF-IDF): the logic behind a tf-idf score is straightforward, words which occur fairly frequently in a body of documents are probably more important, but if they occur too often, then they are too general to be helpful. Mathematically, the tf-idf score for some word t can be described as

$$\text{tf-idf score} = \bar{c_t} \times log(\frac{\text{N}}{n_t})$$

where c_t is the number of times the word t appears in a document, N is the total number of documents, and n is the number of documents in which the word t appears.
We create a term-document matrix. What this means is that on top of having the tf-idf values, each row is a document and each column is a word. If the tweet in row i contains the column in row j, then the element matrix[i][j] will contain the tf-idf value. If the tweet doesn't contain the word, the matrix value will be zero.

Since our feature matrices generated from the count vectorizer and the tf-idf vectorizer are very wide (~15000 columns), we faced a problem of high computational complexity when fitting predictive models. However, we leveraged the sparsity of our feature matrices

and transformed them into a more computationally tractable form called the Compressed Sparse Row representation.

Doc2vec: Doc2vec is an unsupervised algorithm to generate vectors for sentence/paragraphs/documents. The algorithm is an adaptation of word2vec which can generate vectors for words. The vectors generated by doc2vec can be used for tasks like finding similarity between sentences/paragraphs/documents.

Why use Doc2vec?
Bag-of-words models are surprisingly effective, but have several weaknesses.First, they lose all information about word order and second, the model does not attempt to learn the meaning of the underlying words, and as a consequence, the distance between vectors doesn't always reflect the difference in meaning. The Doc2vec method addresses these problems because doc2vec sentence vectors are word order dependent.

For the purpose of this project we have used the Paragraph Vector - Distributed Bag of Words (PV-DBOW) implementation of the model in which the doc-vectors are obtained by training a neural network on the synthetic task of predicting a target word just from the full document's doc-vector. We have combined this with simultaneous training of the word vectors in skip-gram fashion.

We took a 128 dimensional vector for each of the 7613 tweets. Thus, the number of features for each tweet was reduced significantly from 14922 to 128. This is an extremely compressed version of the feature vector as compared to the previous vectorizers (count/tf-idf). In the subsequent sections we compare predictive modelling performance of a number of different models for all the three vectorizers.

III. Modelling Phase

A. Cross Validation Process Overview

The below table gives a high-level overview of the models explored, and the parameter space searched. Whenever possible, we carried out a 'grid search' of the parameter space. Thus, every combination of hyperparameters was tested.

Logistic Regression	Penalty = ['l1', 'l2'], C = [0.01, 0.0464, 0.2154, 1, 4.6416, 21.5443, 100, 464.1589, 2154.4347, 10000]
Support Vector Machine	Kernel:('linear', 'rbf'), C: [0.1,1, 10, 100], Gamma: [1,0.1,0.01,0.001]}
K Nearest Neighbors	n_neighbors: [2,5,10,20,50], weights: ['uniform', 'distance']}
XGBoost	Learning rate: [0.1,0.3, 0.5], #so called `eta` value max_depth: [5, 10, 20], n_estimators: [20, 100, 500], #number of trees, 1000 gives better results}
Neural Network	hidden_layer_sizes: [(5,), (20,),(100,)], activation: ['relu', 'logistic'], alpha: [0.0001, 0.01, 1]
Random Forest	m: [122:292, step =10] m: [11:91, step =10]

B. Model Comparison using 10-fold Cross Validation

Our task here is to do binary classification. We chose to compare seven different types of models on their classification performance.

As an overview, the seven models are:

- Logistic Regression
- Support Vector Machine
- k-Nearest Neighbors
- XG Boost
- Single Layer Neural Net
- Random Forest
- Recurrent Neural Net (LSTM)

In addition, we had to make a decision about what form of vectorization to use for each model. The following vectorization techniques were considered:

- Count Vectorization
- tf-idf vectorization
- word2vec vectorization

The seventh model Recurrent Neural Net (LSTM) we fit didn't require prior vectorization of the tweets as it had inbuilt mechanism to create embeddings from text. We did separate 10-fold CV for this model and finally compiled results of all our models together to find out the best one (minimum mean CV misclassification error).

The models are briefly explained along with their hyperparameters in the following text:

(i) Logistic Regression

Logistic regression is the regression analysis technique used when the dependent variable is dichotomous (binary). At the center of logistic regression analysis is the task of estimating the log odds of an event. We assume a linear relationship between the predictor variables and the log-odds of the event that Y=1. This linear relationship can be written in the following mathematical form (where ℓ is the log-odds, and w_i are parameters of the model):

$$\ell = log(\frac{p}{1-p}) = w_0 + w_1x_1 + w_2x_2 + \ldots + w_ix_i + \ldots + w_mx_m$$

As an optimization problem, binary class $\ell 2$ penalized logistic regression minimizes the following cost function:

$$\min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$$

Similarly, $\ell 1$ regularized logistic regression solves the following optimization problem:

$$\min_{w,c} \|w\|_1 + C\sum_{i=1}^{n} \log(\exp(-y_i(X_i^T w + c)) + 1)$$

We have tuned the hyperparameter of 'penalty' ('ℓ2' or 'ℓ1') along with 10 different values of the hyperparameter 'C' (1.0e-02 to 1.0e+04) using 10 replicates of 10-fold cross validation to obtain the model with the lowest average CV misclassification rate.

In order to find the best combination of hyperparameters, we had to fit 20 (2x10 for 2 'penalty' types and '10' different C values) models for each fold of the cross validation. Then we had to repeat this for 10 replicates of CV (10 different random seeds for generating CV folds). Also, we had to do this for all the three different vectorizers. In total for logistic regression alone, we fit 20x10x10x3 = 6000 models. That could take a lot of time, but since we used the CSR representation of our matrices, the models were fit in a reasonable time.

The complete cross validation results may be referenced from the appendix and only final results are included here for brevity.

The best model appears to be the one where the corpus is **count-vectorized, with a C tradeoff parameter of 0.2154, and an L2 loss.**

(ii) Support Vector Machine (SVM) :

Support-vector machines (SVMs, also support-vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier.

More formally, a support-vector machine constructs a hyperplane or set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks like outliers detection.Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class (so-called functional margin), since in general the larger the margin, the lower the generalization error of the classifier.

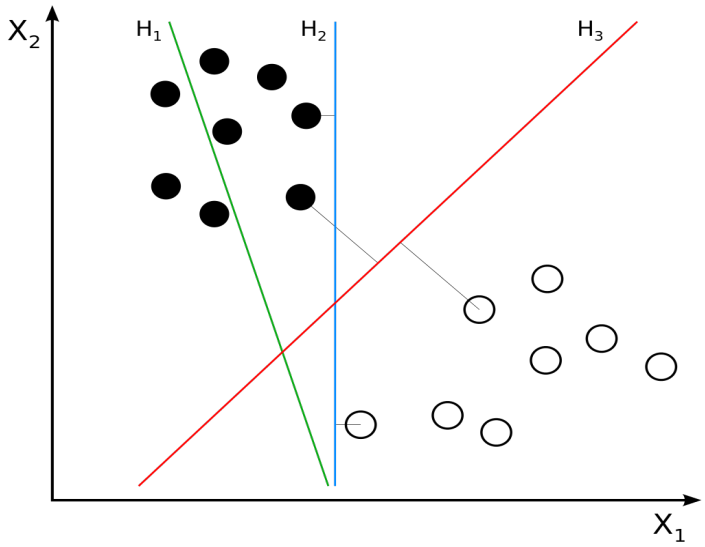

Fig 5 : H1 does not separate the classes. H2 does, but only with a small margin. H3 separates them with the maximal margin.

We used SVM tuning the parameters C and gamma which controls the maximum error allowed (Soft Classification) and influence of an observation in the radial kernel setting respectively. sklearn library implementation of SVM has been used. It is important to note that larger the C and gamma less will be the bias but a higher variance and therefore chances are that the model is overfitting.

The optimal SVM hyperparameters are: C = 1, gamma = 0.1, kernel = 'rbf'

(iii) K-Nearest Neighbors (KNN)

The k-nearest neighbors (KNN) algorithm is a simple, easy-to-implement supervised machine learning algorithm that can be used to solve both classification and regression problems. KNN captures the idea of similarity and hinges on the assumption that similar things exist in close proximity. In this implementation we have compared 5 different values for nearest neighbors and two different weightages namely uniform and distance on each nearest neighbor.Both were performed under cross validation. The objective was to minimize the euclidean distance between the new observation and it's nearest neighbors.

$$\text{Euclidean distance} = \sqrt{\sum_{i=1}^{n} \left(x_i - y_i\right)^2}$$

The optimal k-NN hyperparameters are: a distance based weight, and k = 10.

(iv) XGBoost

XGBoost is an optimized distributed gradient boosting Python library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way.

The optimal parameters were learning rate = 0.1, max_depth = 5, n_estimators = 500.

(v) Single Hidden Layer Neural Network

We used scikit learn's MLP Classifier to train and cross validate a single layer neural network model. Some choices we had to decide between were the choice of activation function, alpha parameter (to reduce overfitting), and number of neurons in the hidden layer.

The optimal parameters were: activation function = relu,alpha = 1, hidden layer size = 5. The appropriate vectorization was: tf-idf vectorization.

(vi) Random Forests

We recognize that Random Forests are a widely used nonparametric modelling technique, hence we decided to include it here as well for completeness.

Our rationale behind carrying out a basic hyperparameter tuning (with only one parameter) for the random forest was that a grid search CV on all hyperparameters would take a long time.

Random Forest is an ensemble method that applies bagging (bootstrap aggregation) to trees, with a small tweak. The number of variables that each tree is allowed to consider when making a split (m) is sampled randomly from the entire set of predictors (p). m is usually less than p for random forests. In practice, m is often chosen as the square root of p. However, for the purpose of the project, we decided to tune the 'm' parameter of RF. To keep training times manageable, we limited each tree size to a depth of 10. The rationale behind this was that max_depth = 10 was identified in the XGBoost model as a reasonably good value for the max-depth.

The optimal parameters were:

- count_vectorizer: m= 272, accuracy = 0.6764
- tf-idf vectorizer: m = 272, accuracy = 0.6696
- doc2vec: m = 31, accuracy = 0.7699

Our final recommended random forest model is: vectorizer: doc2vec, m = 31, to give a maximal CV accuracy of 0.7699.

(vii) RNN (Recurrent Neural Network)

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Derived from feedforward neural networks, RNNs can use their internal state (memory) to process variable length sequences of inputs. For the implementation Keras framework has been used. Keras framework provides an inbuilt embedding layer to perform the encoding of the textual data after

having it being converted to tokens. A tokenizer from nltk library has been used to do the same. The maximum number of words a tweet had in the entire set was 23. Corresponding to this value padding was done in the remaining tweet after tokenizing to have the input layer have coherent size. The following architecture was used

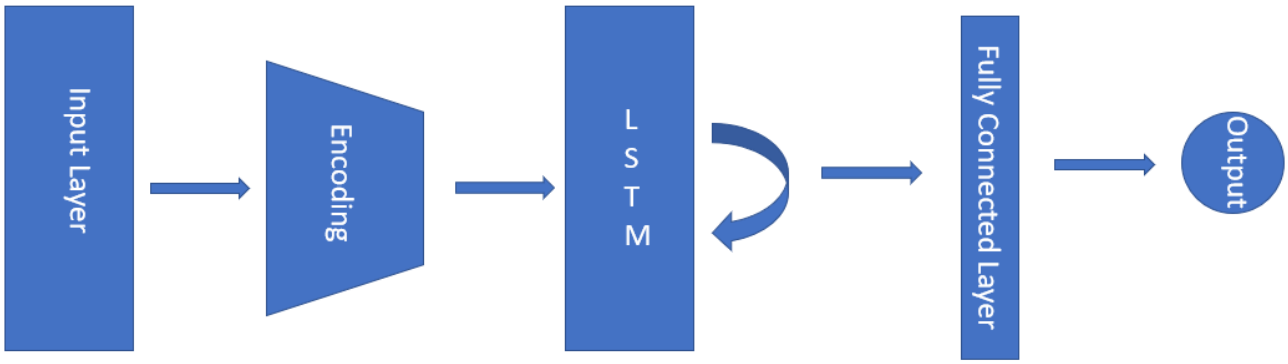

Fig 6 :RNN architecture used

Using 10 fold cross validation on the same folds epochs and learning rate was tuned. It was found the learning rate of 0.0001 and an epoch count of 13 was optimal. The cross validation misclassification rate observed was 0.16728. The batch size was kept constant at 128 and RMSprop was used as the optimiser. Learning rate decay (step decay and exponential decay) was also explored, however no significant improvement over the cross validation tuned results was observed. Below is the training and test loss plotted as a function of epoch for the tuned parameters.

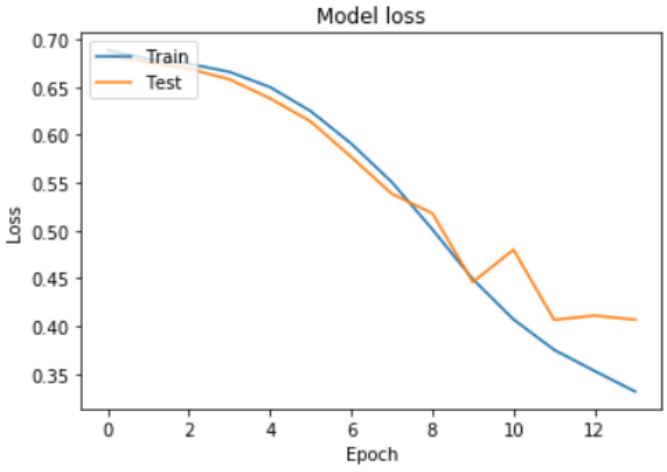

Fig 7 : Train test loss for RNN

The optimum values of hyperparameters tuned for RNN is epoch = 13 and learning rate of 0.0001.

C. Tabulation of Final Results

As a summary of the modelling phase, we include a tabulation of the optimal parameters identified for each of the three vectorizors we used along with the core models evaluated.

(i) Count Vectorizer

Logistic Regression	Score = 0.7986	Best Penalty: l2
	Best C: 0.2154	
Support Vector Machine	Score = 0.8048	Best kernel: rbf
	Best C: 1	Best gamma: 0.1
K Nearest Neighbors	Score = 0.7239	Best n_neighbors: 2
	Best weights: distance	
XGBoost	Score = 0.7930	Best learning rate: 0.1
	Best max_depth: 5	Best n_estimators: 500
Neural Network	Score = 0.8010	Best hidden_layer_sizes: (5,)
	Best alpha: 1	Best activation: relu
Random Forest	Score = 0.6764	Best m: 272

Table 1 : Performance on Count vectorizer

(ii) Tf-Idf

Logistic Regression	Score = 0.7974	Best Penalty: l2
	Best C: 1.0	
Support Vector Machine	Score = 0.8011	Best kernel: rbf
	Best C: 1	Best gamma: 1
K Nearest Neighbors	Score = 0.7869	Best n_neighbors: 20
	Best weights: distance	
XGBoost	Score = 0.7841	Best learning rate: 0.1
	Best max_depth: 5	Best n_estimators: 500
Neural Network	Score = 0.8019	Best hidden_layer_sizes: (100,)
	Best alpha: 1	Best activation: relu
Random Forest	Score = 0.6696	Best m: 272

Table 2 : Performance on tf-idf vectorizer

(iii) Doc2vec

Logistic Regression	Score = 0.7718	Best Penalty: l1
	Best C: 1.0	
Support Vector Machine	Score = 0.7990	Best kernel: rbf
	Best C: 1	Best gamma: 0.1
K Nearest Neighbors	Score = 0.7927	Best n_neighbors: 10
	Best weights: distance	
XGBoost	Score = 0.7935	Best learning rate: 0.1
	Best max_depth: 5	Best n_estimators: 500
Neural Network	Score = 0.7839	Best hidden_layer_sizes: (20,)
	Best alpha: 0.0001	Best activation: relu
Random Forest	Score = 0.7699	Best m: 31

Table 3 : Performance on Dov2vec

The above results have been visualized in Figure 8. The metric used was cross validated accuracy.

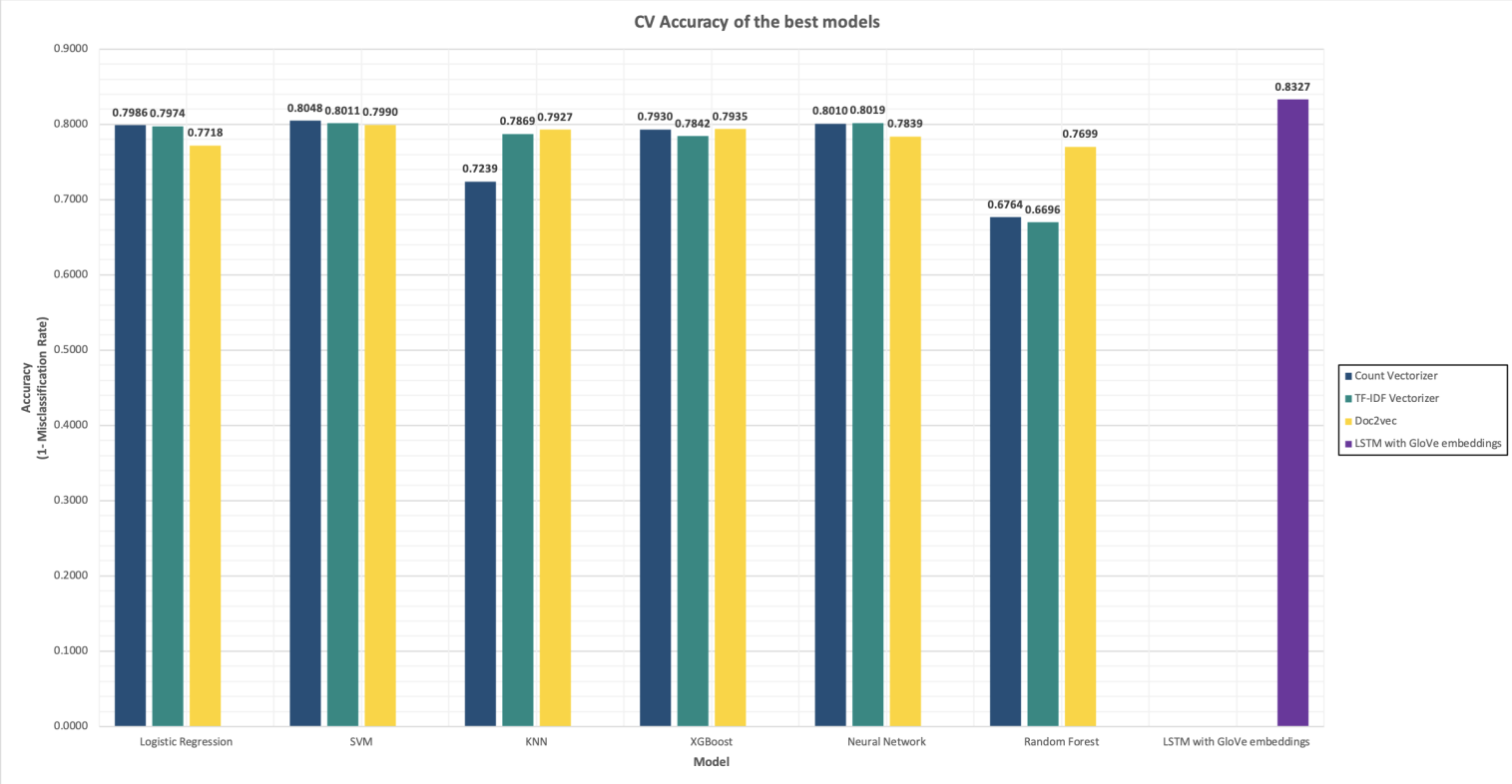

Fig 8 : Performance of algorithms on the three vectorizing method along with LSTM

IV. Conclusion & Further Steps

Our final recommended models are LSTM with Glove Embeddings and Support Vector Machine with Count Vectorization. Each has an accuracy of 83.37% and 80.48% respectively. The optimal parameters may be referenced from the Appendix Section.

With more time, we would carry out the following steps for each of the below phases:

-> text vectorization phase:

- Explore alternative text vectorization approaches

-> data preprocessing phase:

- Feature engineering to produce more variables:
    - tweet sentiment
    - geo-location of tweet
    - time of tweet

-> modelling phase

- alternative modelling approaches with more exigent computational costs (e.g. nnets with more sophisticated architectures, more expansive grid search, etc.)

V. Appendix

A. Cross Validation Results

(i) Logistic Regression

C	penalty	count_vec: mean CV Misclassification error	tf_idf: mean CV Misclassification error	doc2vec: mean CV Misclassification error
0.01	l1	0.4087	0.4298	0.3078
	l2	0.2131	0.2088	0.2402
0.046415888336127774	l1	0.2972	0.4298	0.2398
	l2	0.2069	0.2062	0.23
0.21544346900318834	l1	0.229	0.2795	0.2332
	l2	0.2014	0.2038	0.2298
1.0	l1	0.2108	0.219	0.2282
	l2	0.205	0.2026	0.2297
4.6415888336127775	l1	0.2216	0.217	0.2308
	l2	0.2137	0.206	0.23
21.54434690031882	l1	0.2292	0.2257	0.2295
	l2	0.2249	0.2162	0.2294
100.0	l1	0.2334	0.2289	0.23
	l2	0.2292	0.2255	0.23
464.1588833612773	l1	0.2372	0.2321	0.2301
	l2	0.2335	0.2308	0.2301
2154.4346900318824	l1	0.2456	0.2344	0.2301
	l2	0.2363	0.2344	0.2301
10000.0	l1	0.2686	0.248	0.2301
	l2	0.2381	0.2363	0.2301

(ii) Support Vector Machine

C	gamma	kernel	count_vec: mean CV Misclassification error	tf_idf: mean CV Misclassification error	doc2vec: mean CV Misclassification error
0.1	0.001	linear	0.201	0.2122	0.2241
		rbf	0.5418	0.5418	0.5415
	0.01	linear	0.201	0.2122	0.2241
		rbf	0.314	0.5418	0.2396
	0.1	linear	0.201	0.2122	0.2241
		rbf	0.2867	0.3951	0.2213
	1.0	linear	0.201	0.2122	0.2241
		rbf	0.3969	0.3261	0.3638
1.0	0.001	linear	0.2259	0.2042	0.2248
		rbf	0.2596	0.5418	0.2385
	0.01	linear	0.2259	0.2042	0.2248
		rbf	0.2097	0.2416	0.2267
	0.1	linear	0.2259	0.2042	0.2248
		rbf	0.1953	0.2031	0.201
	1.0	linear	0.2259	0.2042	0.2248
		rbf	0.3495	0.1989	0.2035
10.0	0.001	linear	0.2402	0.2339	0.2262
		rbf	0.205	0.2392	0.228
	0.01	linear	0.2402	0.2339	0.2262
		rbf	0.2042	0.2009	0.2157
	0.1	linear	0.2402	0.2339	0.2262
		rbf	0.2052	0.2113	0.2091
	1.0	linear	0.2402	0.2339	0.2262
		rbf	0.3495	0.2038	0.2178
100.0	0.001	linear	0.2533	0.2468	0.2264
		rbf	0.2061	0.201	0.223
	0.01	linear	0.2533	0.2468	0.2264
		rbf	0.2261	0.2146	0.2028
	0.1	linear	0.2533	0.2468	0.2264
		rbf	0.2108	0.2279	0.2401
	1.0	linear	0.2533	0.2468	0.2264
		rbf	0.3495	0.2039	0.231

(iii) K-NN Classifier

n_neighbors	weights	count_vec: mean CV Misclassification error	tf_idf: mean CV Misclassification error	doc2vec: mean CV Misclassification error
2.0	distance	0.2761	0.263	0.2633
	uniform	0.2942	0.2423	0.2326
5.0	distance	0.2883	0.2332	0.2177
	uniform	0.2969	0.2345	0.2184
10.0	distance	0.2987	0.2204	0.2072
	uniform	0.3307	0.2238	0.2165
20.0	distance	0.3224	0.2131	0.2079
	uniform	0.3592	0.2161	0.2208
50.0	distance	0.3486	0.2139	0.2239
	uniform	0.4051	0.2221	0.2349

(iv) Boosting

learning_rate	max_depth	n_estimators	count_vec: mean CV Misclassification error	tf_idf: mean CV Misclassification error	doc2vec: mean CV Misclassification error
0.1	5.0	20.0	0.2827	0.2912	0.2307
		100.0	0.2377	0.2435	0.2136
		500.0	0.2069	0.2159	0.2066
	10.0	20.0	0.2637	0.2678	0.2212
		100.0	0.2211	0.2295	0.2077
		500.0	0.2074	0.2168	0.2079
	20.0	20.0	0.2439	0.2489	0.2328
		100.0	0.2086	0.2179	0.214
		500.0	0.2139	0.2218	0.2101
0.3	5.0	20.0	0.2503	0.2505	0.2295
		100.0	0.2155	0.2203	0.217
		500.0	0.2128	0.2175	0.2089
	10.0	20.0	0.2392	0.2432	0.2244
		100.0	0.2075	0.2181	0.214
		500.0	0.2205	0.2302	0.2118
	20.0	20.0	0.2168	0.2268	0.2287
		100.0	0.2111	0.2228	0.2128
		500.0	0.2202	0.2354	0.2096
0.5	5.0	20.0	0.2353	0.2407	0.2379
		100.0	0.2086	0.221	0.2217
		500.0	0.2171	0.2283	0.2163
	10.0	20.0	0.2232	0.2308	0.2333
		100.0	0.2132	0.2191	0.2196
		500.0	0.2221	0.2328	0.217
	20.0	20.0	0.2117	0.222	0.2251
		100.0	0.2137	0.2261	0.2146
		500.0	0.2235	0.2381	0.2151

(v) Single Hidden Layer Neural Network

activation	alpha	hidden_layer_sizes	count_vec: mean CV Misclassification error	tf_idf: mean CV Misclassification error	doc2vec: mean CV Misclassification error
logistic	0.0001	100	0.2393	0.2394	0.2173
		20	0.2459	0.24	0.2195
		5	0.2399	0.2355	0.2216
	0.01	100	0.2235	0.2167	0.226
		20	0.2218	0.2135	0.2204
		5	0.2181	0.2103	0.2213
	1	100	0.2165	0.4298	0.2317
		20	0.2092	0.4298	0.2267
		5	0.2101	0.4298	0.2272
relu	0.0001	100	0.2413	0.246	0.2326
		20	0.2452	0.2497	0.2163
		5	0.2505	0.2484	0.2189
	0.01	100	0.225	0.2253	0.2289
		20	0.2283	0.2281	0.2173
		5	0.2287	0.2269	0.2183
	1	100	0.2008	0.1982	0.2175
		20	0.1997	0.1989	0.2217
		5	0.1989	0.1997	0.2278

(vi) Random Forest

m	count_vec: mean CV Misclassification error	tf_idf: mean CV Misclassification error
122	0.354308	0.360094
132	0.351943	0.347873
142	0.346295	0.348265
152	0.345899	0.348005
162	0.343931	0.343405
172	0.341303	0.341696
182	0.336968	0.343928
192	0.335654	0.337357
202	0.339069	0.339201
212	0.342617	0.339204
222	0.328563	0.332895
232	0.329479	0.338542
242	0.331977	0.332107
252	0.329608	0.33434
262	0.32764	0.331192
272	0.323567	0.3304
282	0.328296	0.330925
292	0.331975	0.331581

m	doc2vec: mean CV Misclassification error
11	0.231611
21	0.231743
31	0.230035
41	0.235159
51	0.233058
61	0.235289
71	0.232003
81	0.233449
91	0.231612

(vi) LSTM (RNN)

LR	Epochs	Mean CV Misclassification Error
1e-06	10	0.2768
	11	0.271
	12	0.279
	13	0.2901
	14	0.2849
	15	0.2753
	16	0.2591
	17	0.2537
	18	0.2413
	19	0.2587
	20	0.2561
1e-05	10	0.2652
	11	0.2603
	12	0.2455
	13	0.2222
	14	0.2589
	15	0.2265
	16	0.2345
	17	0.2159
	18	0.2086
	19	0.2147
	20	0.2172
0.0001	10	0.2114
	11	0.202
	12	0.1887
	13	0.1716
	14	0.1673
	15	0.1848
	16	0.2132
	17	0.1933
	18	0.2108
	19	0.2259
	20	0.2372

LR	Epochs	Mean CV Misclassification Error
0.001	10	0.2397
	11	0.2185
	12	0.2123
	13	0.2345
	14	0.2514
	15	0.2426
	16	0.2252
	17	0.2611
	18	0.2544
	19	0.2541
	20	0.2657
0.010	10	0.2629
	11	0.2683
	12	0.2536
	13	0.2768
	14	0.2995
	15	0.3120
	16	0.2982
	17	0.2749
	18	0.2876
	19	0.2793
	20	0.3241

B. *Python Code*

Preprocessing

```
### preprocessing
import re
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import wordnet

def remove_URL(text):
    url = re.compile(r'https?://\S+|www\.\S+')
    return url.sub(r'',text)
df['text']=df['text'].apply(lambda x : remove_URL(x))

def remove_punct(text):
    table=str.maketrans('','',string.punctuation)
    return text.translate(table)
df['text']=df['text'].apply(lambda x : remove_punct(x))

# lemmatization

def lemmatize_sentence(sentence):
    #tokenize the sentence and find the POS tag for each token
    nltk_tagged = nltk.pos_tag(nltk.word_tokenize(sentence))
    #tuple of (token, wordnet_tag)
    wordnet_tagged = map(lambda x: (x[0], nltk_tag_to_wordnet_tag(x[1])), nltk_tagged)
    lemmatized_sentence = []
    for word, tag in wordnet_tagged:
        if tag is None:
            #if there is no available tag, append the token as is
            lemmatized_sentence.append(word)
        else:
            #else use the tag to lemmatize the token
            lemmatized_sentence.append(lemmatizer.lemmatize(word, tag))
    return " ".join(lemmatized_sentence)

# Import stopwords with nltk.
from nltk.corpus import stopwords
stop = stopwords.words('english')
df['lemmatize_without_stopwords'] = df['Lemmatize'].apply(
    lambda x: ' '.join([word for word in x.split() if word not in (stop)]))
```

Vectorizers

```
### count vectorizer

from sklearn import feature_extraction

count_vectorizer = feature_extraction.text.CountVectorizer()
train_vectors = count_vectorizer.fit_transform(input_data["lemmatize_without_stopwords"])
train_matrix = train_vectors.todense()
countvec_train = pd.DataFrame(train_matrix)

### tf-idf vectorizer

from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(use_idf=True)
tfidf_vectorizer_vectors=tfidf_vectorizer.fit_transform(input_data["lemmatize_without_stopwords"])
tfidf_train_matrix = tfidf_vectorizer_vectors.todense()
tfidf_train = pd.DataFrame(tfidf_train_matrix)

### doc2vec vectorizer

import gensim

def read_corpus(dframe, tokens_only=False):
    f = list(dframe['lemmatize_without_stopwords'])
    for i, line in enumerate(f):
        if i != 7626:
            tokens = gensim.utils.simple_preprocess(line)
            if tokens_only:
                yield tokens
            else:
                # For training data, add tags
                yield gensim.models.doc2vec.TaggedDocument(tokens, [i])

train_corpus = list(read_corpus(input_data))

model = gensim.models.doc2vec.Doc2Vec(dm = 0, dbow_words = 1, vector_size=128, min_count=2, epochs=40)
model.build_vocab(full_corpus)
model.train(full_corpus, total_examples=model.corpus_count, epochs=model.epochs)

doc2vec_train = []
for i in range(len(train_corpus)):
    doc2vec_train.append(model.docvecs[i])
```

Cross Validation Framework

```
### Cross validation framework

from sklearn.model_selection import StratifiedKFold
from scipy.sparse import csr_matrix

X = train.iloc[:,:-1]
y = train.iloc[:,-1]

for seed in range(10):
    skf = StratifiedKFold(n_splits=10, random_state=seed[i], shuffle=True)
    skf.get_n_splits(X, y)

    for train_index, test_index in skf.split(X, y):
        print("TRAIN:", train_index, "TEST:", test_index)
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

    folds = [0]*7612
    for i in range(10):
        train_index, test_index = list(skf.split(X, y))[i]
        for idx in test_index:
            folds[idx] = i+1
    train["fold"] = folds
```

Compressed Sparse Matrix (CSR Matrix)

```
# sparcity
X_csr = csr_matrix(X)
```

Models

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn import svm
import xgboost as xgb
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier

### Logistic Regression
logreg = LogisticRegression(solver = 'liblinear', class_weight = 'balanced')
hyperparameters = {'penalty': ['l1', 'l2'], 'C': np.logspace(-2, 4, 10)}
# Create grid search using 10-fold cross validation
clf = GridSearchCV(logreg, hyperparameters, cv=skf, scoring = 'accuracy', verbose=3)
best_fit = clf.fit(X_csr, y)

### SVM
svc = svm.SVC(cache_size = 1000, class_weight = 'balanced', gamma="auto")
hyperparameters = {'kernel':('linear', 'rbf'), 'C': [0.1,1, 10, 100], 'gamma': [1,0.1,0.01,0.001]}
# Create grid search using 10-fold cross validation
clf = GridSearchCV(svc, hyperparameters, cv=skf, scoring = 'accuracy', verbose=3)
best_fit = clf.fit(X_csr, y)

### XGBoost
xgb_model = xgb.XGBClassifier(objective = 'binary:logistic',scale_pos_weight = 1.3, silent = 1, seed = 786)

hyperparameters = {'learning_rate': [0.1,0.3, 0.5], #so called `eta` value
              'max_depth': [5, 10, 20],
              'n_estimators': [20, 100, 500], #number of trees, change it to 1000 for better results}
# Create grid search using 10-fold cross validation
clf = GridSearchCV(xgb_model, hyperparameters, cv=skf, scoring = 'accuracy', verbose=3)
best_fit = clf.fit(X_csr, y)

### KNN
neigh = KNeighborsClassifier()
hyperparameters = {'n_neighbors': [2,5,10,20,50], #so called `eta` value
              'weights': ['uniform', 'distance']}
# Create grid search using 10-fold cross validation
clf = GridSearchCV(neigh, hyperparameters, cv=skf, scoring = 'accuracy', verbose=3)
best_fit = clf.fit(X_csr, y)

### Neural Network
classifier = MLPClassifier(random_state=786)
hyperparameters = {'hidden_layer_sizes': [(5,), (20,),(100,)], #so called `eta` value
              'activation': ['relu', 'logistic'],
              'alpha': [0.0001, 0.01, 1]}
clf = GridSearchCV(classifier, hyperparameters, cv=skf, scoring = 'accuracy', verbose=3)
best_fit = clf.fit(X_csr, y)

### Random Forest
rf = RandomForestClassifier(max_depth=2, random_state = 0)
hyperparameters = {'max_features': np.arange(int(math.sqrt(len(tfidf.columns))), 300, 10)}
clf = GridSearchCV(rf, hyperparameters, cv=skf, scoring = 'accuracy', verbose=3)
best_fit = clf.fit(X_csr, y)
```

RNN (LSTM)

Git Link : https://github.com/manish2020-iitr/Msia2020/blob/master/PA2_LSTM.ipynb

VI. References

- NLTK WordNet Lemmatizer
  https://www.nltk.org/_modules/nltk/stem/wordnet.html
- sklearn feature selection
  https://scikit-learn.org/stable/modules/feature_selection.html
- Gensim Doc2Vec
  https://radimrehurek.com/gensim/auto_examples/tutorials/run_doc2vec_lee.html
- Scipy CSR Matrices
  https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html
- sklearn GridSearchCV https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
- sklearn Logistic Regression https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- sklearn SVM Classifier
  https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
- sklearn KNN Classifier https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
- sklearn XGBoost
  https://xgboost.readthedocs.io/en/latest/python/python_api.html
- sklearn MLP Classifier (Single Hidden Layer Neural Network)
  https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
- sklearn Random Forest Classifier
  https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html