
Real or Not? NLP with Disaster Tweets

Gryffindor

Anit Gupta

Masters in Data Science
Georgia State University
agupta33@student.gsu.edu

Nikhil Gupta

Masters in Data Science
Georgia State University
ngupta9@student.gsu.edu

Sameera Turupu

Masters in Computer Science
Georgia State University
sturupu1@student.gsu.edu

Saurabh Shrinivas Maydeo

Masters in Computer Science
Georgia State University
smaydeo1@student.gsu.edu

Sonam Dawani

Masters in Data Science
Georgia State University
sdawani1@student.gsu.edu

Susanth Dasari

Masters in Computer Science
Georgia State University
sdasari3@student.gsu.edu

Abstract

This project is an attempt to build a binary classification Machine Learning model using different approaches for Natural Language Processing. The methodology discussed in this paper tries to classify a twitter message or 'tweet' into two main classes - 'Real' and 'Non Real'. These classes depict whether a given message is announcing or related to any disaster or not. The techniques discussed and attempted in this project builds an automated pipeline from Data Cleaning to Data Modelling. This is also an attempt to show the applications of modelling techniques like Support Vector Machines(SVM), XGBoost, Long short-term memory(LSTM) and Bidirectional Encoder Representations from Transformers(BERT) in the area of Natural Language processing.

1 Introduction

1.1 Motivation

The use and impact of social media platforms (e.g., Twitter, Facebook, LinkedIn) has skyrocketed over the past decade. They have become critical components of emergency preparedness, response, and recovery. Twitter has been widely regarded as active communication channel during emergency events such as disasters caused by natural hazards. A tweet could even serve as first news regarding an accident or a fire. If these tweets are noticed early by response teams, they would get more time react to the situation. Government bodies are closing monitoring twitter to be informed early about any emergency or a disaster. The motivation of this project comes from an incident that happened in India last year. The Indian Railways system was able to successfully save a person's life by reacting early to a twitter message. They were able to arrange medicines and equipment for a patient during a journey, thereby avoiding an incident. The example serves as learning that if monitor correctly, the social media platforms can act as a source of news concerning risks or tragedies.

The task is not as easy as it looks. On average, there are 6000 new tweets every second. Reading and classification at this rate is impossible for a human. The problem requires the utilization of machines and Natural Language Processing. But, this leads to the question of contextual understanding of a sentence by a computer. It is not always clear that a person's tweet is announcing a disaster or not. A tweet reads, "**Kobe is on fire today**". It is tough for a machine to realize that this tweet is not about a fire but, about a person's game.

The task of this project is to develop a contextual based machine learning model to process natural language. The model will be a part of an automated pipeline to classify disaster and general tweets. The data source for this project is a Kaggle competition[1]. Kaggle is an online community of data scientists and machine learning practitioners.

1.2 Data and Exploration

The raw data that is provided by Kaggle is in the form of Training set and Test set. There are 4 features in the data set which are ID, keyword, location, text. The column 'target' is the class variable.

There are 7613 samples in training set and 3263 entries in test set. There are 33.27 % missing values in location column. Most of the tweets are from "North America" Thus, we can drop this attribute as it is not much of our use.

<pre>Training data shape (rows, cols): (7613, 5) ****Train Data Info**** <class 'pandas.core.frame.DataFrame'> RangeIndex: 7613 entries, 0 to 7612 Data columns (total 5 columns): id 7613 non-null int64 keyword 7552 non-null object location 5080 non-null object text 7613 non-null object target 7613 non-null int64 dtypes: int64(2), object(3) memory usage: 297.5+ KB</pre>	<pre>Test data shape (rows, cols): (3263, 4) ****Test Data Info**** <class 'pandas.core.frame.DataFrame'> RangeIndex: 3263 entries, 0 to 3262 Data columns (total 4 columns): id 3263 non-null int64 keyword 3237 non-null object location 2158 non-null object text 3263 non-null object dtypes: int64(1), object(3) memory usage: 102.1+ KB</pre>
---	--

Figure 1: Training set and Test set info

You can see the distribution of our class variable - 'target' in figure 2(a). Disaster tweet is represented by 1 and non-disaster tweet is represented by 0. Out of the all tweets, around 43 % tweets are disaster tweets and 57% tweets are non-disaster tweets. Thus, there is no class imbalance in the data set.

We analysed some of the top keywords for disaster and non-disaster tweets. You can see the frequency distribution of the top keywords for disaster and non-disaster tweets in figure 2(b). There is % sign at some places. From this we got to know that if we are going to even use this feature, then we need to handle it during the preprocessing phase. There are words such as harm, Armageddon, wrecked which are occurring in non-disaster tweets. This tells us that we cannot rely on this feature. Our model needs to be able to understand the semantic meaning from the tweet itself.

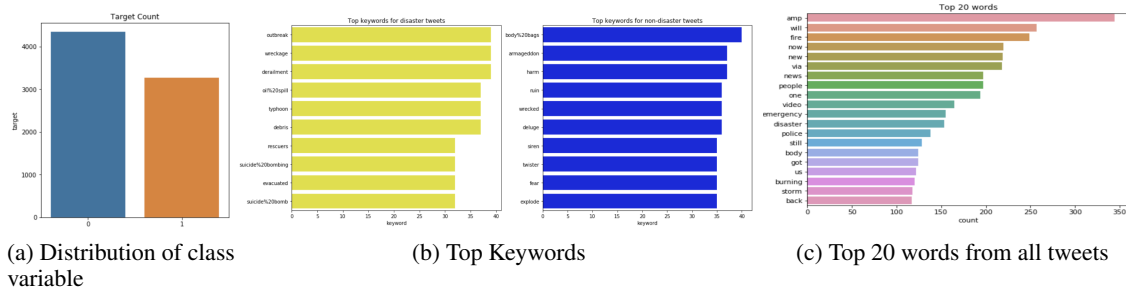
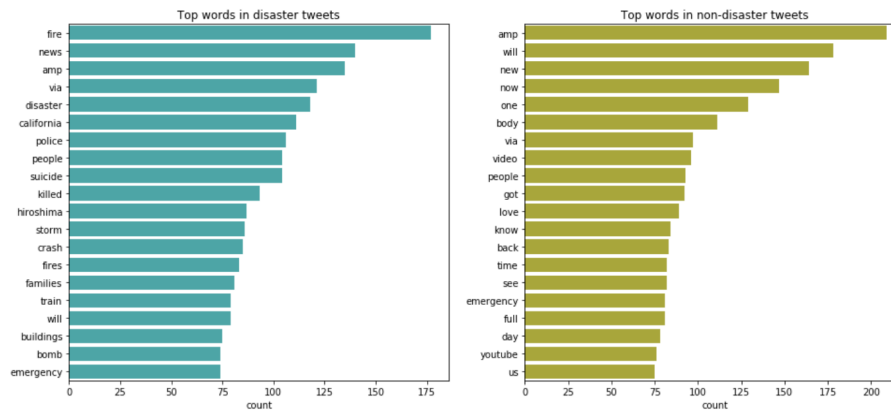


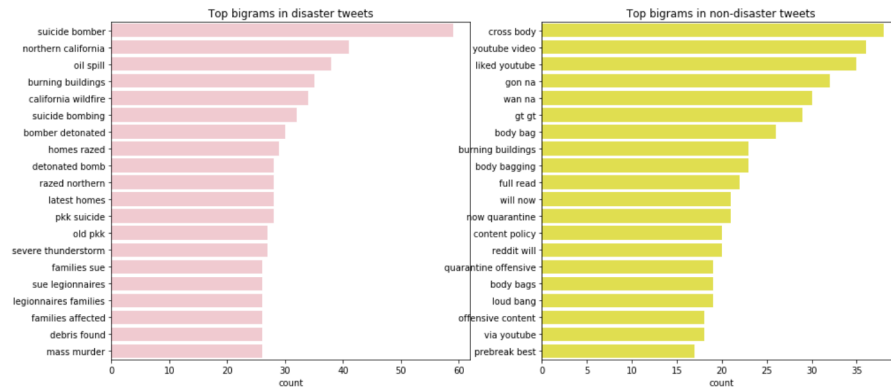
Figure 2

The 'text' feature contains actual tweets which can have misspelled words, abbreviations, hash tags, user mentions, urls. We need to handle these during preprocessing phase. You can see in figure 3 that the average length of a tweet is 88 characters. Highest frequency of tweets is within 136 139 characters. We tried to get insights from 'text' - tweet feature by plotting unigrams, bigrams, etc. For this, we performed the basic preprocessing of 'text' feature. This involves tasks like removal of links, removal of line breaks removal of leading, trailing, and extra spaces, etc. After performing this basic preprocessing we used this preprocessed data while plotting unigrams, bigrams, etc.

In figure 2(c), you can see the top 20 words in all tweets. These are from disaster and non-disaster tweets combined. Words like 'amp', 'fire', 'now' are some of the most frequently occurring words.



(a) Top unigrams



(b) Top bigrams

Figure 3

To be able to understand which words are occurring in disaster tweets and non-disaster tweets separately, we plotted graph representing unigrams and their counts. In figure 3(a) you can see the unigrams for disaster tweets on the left and unigrams for non-disaster tweets on the right. Some of higher frequency words in disastrous tweets are 'death', 'fire', 'amp', 'suicide', 'flood', 'Hiroshimaa'. Some of the top unigrams for non-disaster tweets are 'amp', 'new', 'now', 'will', etc. The unigram 'amp' is occurring very frequently in disaster and non-disaster tweets.

We also plotted some of the top bigrams for disaster and non-disaster tweets as you can see in figure 3(b). As we can see, 'burning building' is one of the top bigrams for non-disaster tweets which seems to be part of a disaster tweet.

These are some of the insights that we got after extensive exploratory data analysis.

2 Data Preprocessing

The major part of data set consists of post made on the social media application Twitter. As we are dealing with social media's unstructured text, we had to do more than usual preprocessing steps. We have developed methods for different text processing tasks. As we have implemented multiple models for the given classification task, relevant methods for processing text are used for different models. The methods implemented and their detail are given below:

- **to_lower** : Converting text to lower case.
This simple method is important because most of the models will perform much better when same words with different cases are treated as same. Also in case of using any word embedding, it can be uncased, meaning the words are represented in lower case in the embedding. In order to use the embedding we need to convert text to lower case.
- **remove_url** URL link to '[URL]' as token.
Rather than removing URL link we replaced it with a token as '[URL]' word. By doing so we remove the noise and still hold the information.
- **remove_punct** Replace punctuation with space. For better model performance the word attached with punctuation should be considered same as simple word. So that its semantic meaning is identified. For example, 'goal!' and 'goal' should be considered same.
- **remove_special_ucchar** Removing characters in HTML name code.
Some characters like ampersand are in their HTML name code in the given tweet text. For example, '&' is given as '&'. These characters were not removed with above method and required special handling.
- **remove_numbers** Removing numbers.
Any number from tweet text is removed.
- **remove_mentions** Removing mentions.
On Tweeter post we can mention any other tweet users by adding '@' at the beginning of the username. This generates link in the post to redirect to the mentioned user's profile page. So if we only remove the punctuation ('@' here) then the username is left which is generally not helpful and sometimes misleading for models performance. Hence we removed user mentions.
- **handle_unicode** Handles Unicode characters.
This method handles the Unicode characters and replace with '?'. For example, `\xea\x80\x80abcd\xde\xb4` is replaced with '???abcd??'.
- **remove_square_bracket** Removing square bracket values.
In our data set we saw there were some garbage values in square brackets. Hence we removed them.
- **remove_angular_bracket** Removing angular bracket values.
Similar to above, there were some garbage values in angular brackets. Hence we removed them.
- **remove_newline** Removing new line.
This method removes characters '\n' and also actual new line.

- `remove_words_with_numbers` Removing word containing number.
Many of slang words today contains numbers, so we are removing words with number. For example, words like '3RZDA' (meaning Thursday) and 'b4' (meaning before) are removed.
- `hashtag_to_words` Splitting hashtag words.
This method is inspired by analyzing our data set, as it contains lots of hashtags. If we pass the hashtag as it is to the model it will be processed as a new word or out of vocabulary word. Hence there will be no significant semantics meaning for the given hashtag. By just simply removing hashtag we would have lost the information stored in the words of the hashtag. Hence instead of just removing hashtag we replaced hashtag with the split words of the hashtag.
For example, #FireNoodlesChallenge will be replaced by 'fire noodles challenge'.

For separator we used 'symspelly' package[12]. We found the hyper-parameter tuning working best for our data set as: `max_edit_distance_dictionary = 2` and `prefix_length = 8`.

We can see the impact of this method. Initially there were 23,337 unique words and after pre-processing the unique word count reduced by around 1000 words. This means that the hashtags were being counted as new words and after splitting the words were found to be already in the vocabulary.

```
Preprocessing step: hashtag_to_words
Unique Char Count ---> Before: 66 | After: 65
Unique Word Count ---> Before: 23337 | After: 22510
```

Figure 4: Unique word count reduction due to hashtag to words.

- `extra_spaces` Removing extra spaces.
- `remove_stopwords` Removing mentions.
Removes the stop words like 'the', 'is' and 'and'.
- `correct_misspelled_with_context` Correcting misspelled words with context.
As the text of our data set is from social media, we can expect misspelled words. Correcting the misspelled words will enhance almost every model's performance. Sometimes the misspelled word have more than one closest correct word, in such cases we want to have the corrected word which suits the context the most.
For correcting misspelled we used 'symspelly' package. We found the hyper-parameter tuning working best for our data set as: `max_dictionary_edit_distance=3`, `prefix_length=7`.
- `stemming_text` Word to its root word.
This method replace the word to its root word. For example, 'running' is replaced by 'run'. As the semantic meaning for a word and its root word is almost the same.
- `lemmatization` Lemmatization of words.
Lemmatization is an alternative approach from stemming to removing inflection. For example, stemming of 'leaves' would give 'leav', whereas lemmatization of 'leaves' is 'leaf'.
- `removeRepeated` Remove repeated characters of a word.
This method is inspired by the text of our data set. We observed many words with repeated characters, which denote excitement or high emotion but to process these text we needed the actual correct word. For example, 'cooooooll' is replaced by 'cooll' and further by `correct_misspelled_with_context` method is correct to 'cool'.
- `Expand_Contractions` Expands contraction.
This method expands all contraction like 'you'll've' to 'you shall have'. Although most of the words after expanding comes under stop words and will be removed, but some models performs better without removing stop words. In such case we can apply this method while pre-processing.

NOTE: These methods are not listed in sequence of application to the data. The sequence and selection of pre-processing steps differs from model to model.

To evaluate our pre-processing steps, we compared our data set vocabulary before and after the every pre-processing step with BERT vocabulary. Coming sections illustrate and explain about BERT in detail. For this section we need to know that BERT can provide embedding for the words and these embeddings hold semantic meaning of the word. So we want that most of our vocabulary words should be present in BERT embeddings.

Pre-processing had a significant impact. As we can see in Figure:5 Before pre-processing only 66% of words in all text were found in BERT embeddings, where as after pre-processing 94% of words in all text were found in BERT embeddings.

Before pre-processing	
Found embeddings for	23.46% of vocab
Found embeddings for	66.31% of all text
After pre-processing	
Found embeddings for	74.92% of vocab
Found embeddings for	93.86% of all text

Figure 5: Before and after pre-processing

Figure:6 provides details about each pre-processing method contribution towards above score. We can see that methods which had significant impact are hashtag_to_words, Expand_Contractions and correct_misspelled_with_context.

For Training data:	
Preprocessing step: handle_unicode	
Unique Char Count ---> Before: 122 After: 94	
Unique Word Count ---> Before: 32017 After: 32000	
Found embeddings for 23.85% of vocab	
Found embeddings for 67.23% of all text	
Preprocessing step: to_lower	
Unique Char Count ---> Before: 94 After: 68	
Unique Word Count ---> Before: 32000 After: 28104	
Found embeddings for 37.41% of vocab	
Found embeddings for 84.56% of all text	
Preprocessing step: remove_newline	
Unique Char Count ---> Before: 68 After: 67	
Unique Word Count ---> Before: 28104 After: 27967	
Found embeddings for 37.43% of vocab	
Found embeddings for 84.57% of all text	
Preprocessing step: remove_url	
Unique Char Count ---> Before: 67 After: 67	
Unique Word Count ---> Before: 27967 After: 23374	
Found embeddings for 46.89% of vocab	
Found embeddings for 86.49% of all text	
Preprocessing step: remove_special_ucchar	
Unique Char Count ---> Before: 67 After: 66	
Unique Word Count ---> Before: 23374 After: 23337	
Found embeddings for 46.89% of vocab	
Found embeddings for 86.35% of all text	
Preprocessing step: hashtag_to_words	
Unique Char Count ---> Before: 66 After: 65	
Unique Word Count ---> Before: 23337 After: 22510	
Found embeddings for 48.04% of vocab	
Found embeddings for 86.48% of all text	
Preprocessing step: remove_mentions	
Unique Char Count ---> Before: 65 After: 64	
Unique Word Count ---> Before: 22510 After: 20185	
Found embeddings for 54.69% of vocab	
Found embeddings for 87.91% of all text	
Preprocessing step: Expand_Contractions	
Unique Char Count ---> Before: 64 After: 65	
Unique Word Count ---> Before: 20185 After: 20088	
Found embeddings for 54.76% of vocab	
Found embeddings for 88.85% of all text	
Preprocessing step: extra_spaces	
Unique Char Count ---> Before: 65 After: 65	
Unique Word Count ---> Before: 20088 After: 20087	
Found embeddings for 54.76% of vocab	
Found embeddings for 88.85% of all text	
Preprocessing step: removeRepeated	
Unique Char Count ---> Before: 65 After: 65	
Unique Word Count ---> Before: 20087 After: 20093	
Found embeddings for 54.74% of vocab	
Found embeddings for 88.83% of all text	
Preprocessing step: correct_misspelled_with_context	
Unique Char Count ---> Before: 65 After: 38	
Unique Word Count ---> Before: 20093 After: 11586	
Found embeddings for 74.92% of vocab	
Found embeddings for 93.86% of all text	
CPU times: user 7min 30s, sys: 292 ms, total: 7min 30s	
Wall time: 7min 49s	

Figure 6: BERT embedding coverage by each pre-processing step

3 Initial Models

3.1 SVM - Baseline Model

Support Vector Machine (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. This model extracts a best possible hyper-plane / line that segregates the two classes. SVM uses a technique called the kernel trick. The kernel takes a low-dimensional input space and transforms it into a higher dimensional space. In other words, it converts non-separable problem to separable problems by adding more dimension to it.

Here, RBF (Radial Basis Function) kernel is used. RBF Kernel is a popular kernel function commonly used in support vector machine classification. RBF can map an input space in infinite dimensional space. γ is a kernel coefficient which ranges from 0 to 1. A higher value of γ will perfectly fit the training dataset, which causes over-fitting. $\gamma=0.1$ is considered to be a good default value. The value of γ needs to be manually specified in the learning algorithm.

Preprocessing steps used for this model are removing newline character, URL, html, emojis, mentions with @, square and angular brackets, repeated words and tweets, expanding contractions and lemmatization. The input to this model is processed text which is converted to matrix of tf-idf features. GridSearchCV from sklearn is used for tuning hyper parameters. After running this function, we got the best value for c as 1.5, γ as 0.6. C is a regularization parameter that controls the trade-off between achieving a low training error and a low testing error that is the ability to generalize your classifier to unseen data.

Model	Test Accuracy	Test F1 Score	Kaggle Score
SVM	0.7974	0.7271	0.7995

3.2 XGBOOST

SVM gave satisfactory results, but while considering informal language sources such as tweets, the data is always with high levels of noise. As the tweets are manually labelled, noise is expected even in labelling of the data. So, a simple decision tree based model, that is equipped with Bagging and Random Forest(to handle high dimensions), Boosting(to put some context into the words from each tweet) and gradient decent to arrive at minimal loss sounds promising.

XGBOOST (Extreme Gradient Boosting) is a powerful decision-tree based Ensemble model that employs Gradient Descent which has a history of performing extremely well in binary classification tasks. It is similar to gradient boosting but more efficient and it has a capacity to do parallel computation on a single machine. The preprocessing steps for this model are handling unicode characters, removing newline character, URL, special characters, mentions with @, words with numbers, punctuations, repeated and stop words, converting hashtags to words, converting text to lower and lemmatization. XGBoost only works with numeric vectors. Therefore, you need to convert all other forms of data into numeric vectors. A simple method to convert a collection of text documents to a matrix of token counts is CountVectorizer from sklearn. It creates vectors that have a dimensionality equal to the size of data vocabulary, and if the text data features that vocab word, one is inputted in the dimension. Every time the word is encountered, count is incremented, leaving 0s everywhere the word never occurred.

Since the model is aided by Gradient Descent; regularization, learning rate and many other such hyper-parameters come into play. optimization of hyper-parameters like max-depth of tree is also needed based on the Occam's razor to keep depth of the tree as shallow as possible to avoid over-fitting.

Model	Test Accuracy	Test F1 Score	Kaggle Score
XGBOOST	0.7787	0.7303	0.8016

3.3 LSTM

Long Short-Term Memory Networks are a particular type of recurrent neural networks capable of learning long term dependencies. It can learn to keep only relevant information to make predictions, and forget non relevant data.

Bidirectional LSTMs are an extension of traditional LSTMs that can improve model performance on sequence classification problems. In problems where all timesteps of the input sequence are available, Bidirectional LSTMs train two instead of one LSTMs on the input sequence. The first on the input sequence as-is and the second on a reversed copy of the input sequence. This can provide additional context to the network and result in faster and even fuller learning on the problem. Bidirectional LSTMs are supported in Keras via the Bidirectional layer wrapper. This wrapper takes a recurrent layer (LSTM) as an argument. It allows you to specify the merge mode which by default is concatenate, that is how the forward and backward outputs should be combined before being passed on to the next layer.

Preprocessing steps for this model are converting text to lower case, removing newline character, html, emoticons, words with numbers, mentions, words with numbers, punctuations, repeated and stop words, converting hashtags to words, converting text to lower and lemmatization. The processed text is vectorized by a tokenizer which transforms input text to sequence of integers (each integer being the index of a token in a dictionary) which is taken as the input to the model. This model has an embedding layer, instead of loading random weight, weights from our glove and crawl embeddings are loaded. This layer encodes the input sequence into a sequence of dense vectors. Recurrent Neural networks like LSTM generally have the problem of overfitting. Dropout is applied between the embedding and LSTM layers using the Dropout Keras layer. Two Bi-Directional LSTM layers with 128 memory units are used. Then, concatenated one directional max and average pooling are added to reduce dimensionality. Three dense layers with sigmoid and relu activations are added. Finally, because this is a binary classification problem, the binary log loss from keras is used. The efficient ADAM optimization algorithm is used to find the weights and the accuracy metric is calculated and reported for each epoch.

Model	Test Accuracy	Test F1 Score	Kaggle Score
LSTM	0.8139	0.7666	0.8026

4 BERT For NLP

4.1 Core idea behind Bert

In the pre-BERT world, a language model would have looked at this text sequence during training from either left-to-right or combined left-to-right and right-to-left. This one-directional approach works well for generating sentences — we can predict the next word, append that to the sequence, then predict the next to next word until we have a complete sentence. But its difficult to predict when it comes to long sentence.

Instead of predicting the next word in a sequence, BERT makes use of a novel technique called Masked LM (MLM): it randomly masks words in the sentence and then it tries to predict them. Masking means that the model looks in both directions and it uses the full context of the sentence, both left and right surroundings, in order to predict the masked word. Unlike the previous language models, it takes both the previous and next tokens into account at the same time. The existing combined left-to-right and right-to-left LSTM based models were missing this “same-time part”. (It might be more accurate to say that BERT is non-directional though.)

4.2 Behind the scene Bert Working

BERT relies on a Transformer (the attention mechanism that learns contextual relationships between words in a text). A basic Transformer consists of an encoder to read the text input and a decoder to produce a prediction for the task. Since BERT’s goal is to generate a language representation model, it only needs the encoder part. The input to the encoder for BERT is a sequence of tokens, which are

first converted into vectors and then processed in the neural network. But before processing can start, BERT needs the input to be massaged and decorated with some extra metadata:

- Token embeddings: A [CLS] token is added to the input word tokens at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
- Segment embeddings: A marker indicating Sentence A or Sentence B is added to each token. This allows the encoder to distinguish between sentences.
- Positional embeddings: A positional embedding is added to each token to indicate its position in the sentence.

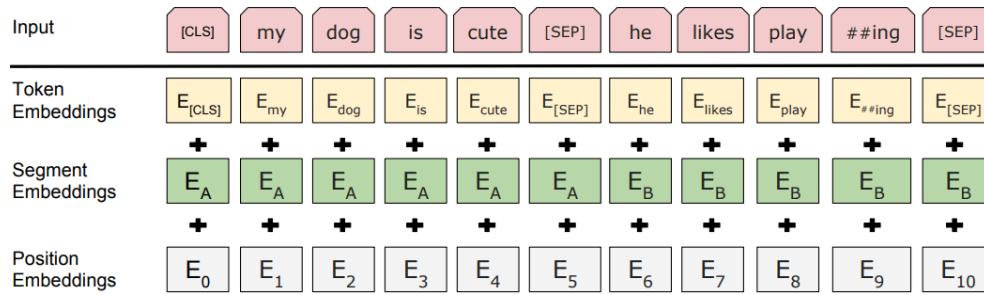


Figure 7: The input representation for BERT: The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

4.3 BERT Architecture

There are four types of pre-trained versions of BERT depending on the scale of the model architecture:

- BERT-Base: 12-layer, 768-hidden-nodes, 12-attention-heads, 110M parameters
- BERT-Large: 24-layer, 1024-hidden-nodes, 16-attention-heads, 340M parameters

For more hyper parameters tuning we can refer the google original paper on BERT at this [Link](#)

4.4 Advanced techniques BERT pre-training

BERT is trained on two language modeling techniques. Masked Language modelling and next sentence prediction. The difference is and what makes BERT stand out is that it is trained on both techniques simultaneously. We will explain the two techniques as they are quite crucial as this helps to understand and decide where we can implement the BERT.

- Masked Language Modeling (MLM) :
The idea here is “simple”: Randomly mask out 15% of the words in the input — replacing them with a [MASK] token — run the entire sequence through the BERT attention based encoder and then predict only the masked words, based on the context provided by the other non-masked words in the sequence. However, there is a problem with this naive masking approach — the model only tries to predict when the [MASK] token is present in the input, while we want the model to try to predict the correct tokens regardless of what token is present in the input. To deal with this issue, out of the 15% of the tokens selected for masking:

80% of the tokens are actually replaced with the token [MASK].

10% of the time tokens are replaced with a random token.

10% of the time tokens are left unchanged.

While training the BERT loss function considers only the prediction of the masked tokens and ignores the prediction of the non-masked ones. This results in a model that converges much more slowly than left-to-right or right-to-left models.

- **Next Sentence Prediction:**
In order to understand relationship between two sentences, BERT training process also uses next sentence prediction. A pre-trained model with this kind of understanding is relevant for tasks like question answering. During training the model gets as input pairs of sentences and it learns to predict if the second sentence is the next sentence in the original text as well.

As we have seen earlier, BERT separates sentences with a special [SEP] token. During training the model is fed with two input sentences at a time such that:

50% of the time the second sentence comes after the first one.
50% of the time it is a random sentence from the full corpus.

BERT is then required to predict whether the second sentence is random or not, with the assumption that the random sentence will be disconnected from the first sentence:

```
Input = [CLS] the man went to [MASK] store [SEP]
        he bought a gallon [MASK] milk [SEP]
Label = IsNext

Input = [CLS] the man [MASK] to the store [SEP]
        penguin [MASK] are flight ##less birds [SEP]
Label = NotNext
```

Figure 8: To predict if the second sentence is connected to the first one or not

4.5 WordPiece Tokenization - BPE

In this approach an out of vocabulary word is progressively split into sub words and the word is then represented by a group of sub words. Since the sub words are part of the vocabulary, we have learned representations a context for these sub words and the context of the word is simply the combination of the context of the sub words. For instance this technique splits token like "playing" to "play" and "##ing". This mainly to cover a wider spectrum of Out-Of-Vocabulary (OOV) words.

5 Model Training and Testing

5.1 Pre-trained BERT

For the purpose of tweet classification we have chosen a pre-trained model of BERT:

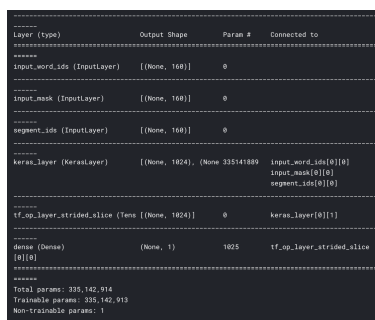
- BERT-Large is of 12 layers, 1024 hidden layers and 16 attentions head, which sums up to a total of 340 Million parameters.
- This model is trained on BooksCorpus dataset and Wikipedia filtered only for English language pages.
- This makes the model trained on a staggering total of 3.3 Billion words.
- This is a uncased English-only model, meaning we cannot use accented letters.
- The final vocabulary for this model is filtered for the top 30,000 words.
- The output of this model is a 1024 dimensional vector, which can be inferred as a embedding for a single tweet containing multiple sentences and words.
- Initially, embeddings for the words are generated, and are passed through the attention blocks to aggregate them into a single vector for the whole tweet.

These characteristics of the chosen pre-trained model, makes it a good fit for us to work with.

5.2 Vanilla BERT

5.2.1 Model Architecture and Training

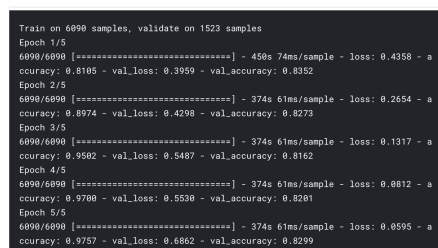
Our initial model implementation using BERT is pretty basic. We are accessing the pre-trained module through Tensorflow hub and adding it as a layer in Keras model. The BERT model takes three inputs as explained earlier, so we have added three input layers for each of the token, segment and position embeddings and pass it to the accessed Tensorflow hub BERT layer. The BERT model added as a layer in our model, outputs 1024 dimension vector for each tweet, this layer is connected to a single neuron Dense layer with **Sigmoid** Activation function for Classification of Disaster and Non-disaster tweets.



```
Layer (type) Output Shape Param # Connected to
-----
input_word_ids (InputLayer) [(None, 100)] 0
input_mask (InputLayer) [(None, 100)] 0
segment_ids (InputLayer) [(None, 100)] 0
keras_layer (KerasLayer) [(None, 1024), (None 33514889 input_word_ids[0][0]
input_mask[0][0]
segment_ids[0][0]
tf_op_layer_strided_slice (TensorFlow) [(None, 1024)] 0 keras_layer[0][1]
dense (Dense) (None, 1) 1025 tf_op_layer_strided_slice[0][0]
Total params: 335,142,914
Trainable params: 335,142,913
Non-trainable params: 1
```

Figure 9: Vanilla BERT model summary

This model uses **Adam** Optimizer with a learning rate of **1e-6**. The reason for using such a low learning rate is, the pre-trained model is trained at a learning rate factor of 10^{-4} , when fine-tuning it's usually recommended that we use a learning rate lower than the the original one.



```
Train on 6890 samples, validate on 1523 samples
Epoch 1/5
6890/6890 [=====] - 450s 74ms/sample - loss: 0.4358 - a
ccuracy: 0.8105 - val_loss: 0.3959 - val_accuracy: 0.8352
Epoch 2/5
6890/6890 [=====] - 374s 61ms/sample - loss: 0.2654 - a
ccuracy: 0.8974 - val_loss: 0.4298 - val_accuracy: 0.8273
Epoch 3/5
6890/6890 [=====] - 374s 61ms/sample - loss: 0.1317 - a
ccuracy: 0.9502 - val_loss: 0.5487 - val_accuracy: 0.8162
Epoch 4/5
6890/6890 [=====] - 374s 61ms/sample - loss: 0.0812 - a
ccuracy: 0.9706 - val_loss: 0.5530 - val_accuracy: 0.8201
Epoch 5/5
6890/6890 [=====] - 374s 61ms/sample - loss: 0.0595 - a
ccuracy: 0.9757 - val_loss: 0.5862 - val_accuracy: 0.8299
```

Figure 10: Vanilla BERT model training

After training for just one epoch, the training loss and accuracy stands at 0.4358 & 0.8105, and the validation loss and accuracy at 0.3959 & 0.8352. Just after the second epoch we can see that the validation loss increases with a decrease in validation accuracy. This is a pointer that the model is over-fitting.

5.2.2 Challenges

Just after one epoch the model starts to overfit, even with a very low learning rate of factor 10^{-6} . We pushed the learning rate to as low as it goes where the training still takes place and it still overfits just after one epoch. The main reasons for this behaviour could be:

- Complex model architecture of BERT.
- Less training data.
- The number of parameters that are updated after each batch in a single epoch.
- The size of the model also leads to a limited batch size(8/16/32). Anything more than that usually doesn't work.
- The limitation of batch size also increases the number of weight updations performed in a single epoch, which is another reason for over-fitting.

5.2.3 Results

The final three layers of BERT are fully connected layers and they do a fine job of classification by themselves. But the above mentioned challenges along with the fact that the model outputs it's best results with only a single-epoch of training, makes the model very unstable and susceptible to the initialization of weights.

Nevertheless, we got our best Kaggle leaderboard scores with this model. Kaggle leaderboard scores are calculated on 30% unrevealed partition of Test data.

Model	Test Accuracy	Test F1 Score	Kaggle Score
Vanilla BERT	0.8461	0.8086	0.8548

5.3 Improved model

We discussed a basic model using BERT and the challenges that came with it. We improved the model in lieu of layers and parameters to address these issues. We also implemented some Training strategies for the same reasons.

5.3.1 Improvements

First and foremost, we aimed at slowing down the convergence of model. After the BERT layer from the [section 5.2.1], we added **three Dense layers** of size 1024 neurons with **ReLU** Activation functions and a **L2 Regularization** of **1e-5** on the Activity of these layer. To this we connected a Single Neuron Dense layer with **Sigmoid** Activation for classification.

These subtle changes to the model were successful in slowing down the convergence and giving us a little more stable results, but we were able to run only 2-3 epochs without over-fitting.

5.3.2 Training Strategy 1

The main motivation for this training strategy is to make less updations of weight before decreasing the learning rate. The straightforward way of doing this is by increasing the batch size of an epoch, but with BERT we have limitations on batch size. So we came up with a manual shuffling and splitting of training data for getting the effect.

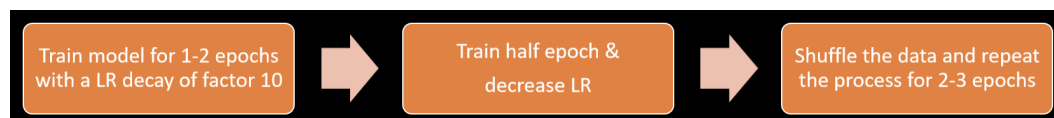


Figure 11: Training Strategy 1

By implementing this strategy, we were able to train the model for more epochs which started giving us more stable results. We were also able to decrease the susceptibility of weight initialization over final results. Finally our best performing model with this strategy looks like below.

Model	Test Accuracy	Test F1 Score	Kaggle Score
BERT + Training Strategy 1	0.8464	0.8082	0.8527

5.3.3 Training Strategy 2

Training strategy 2 is a classic fine tuning method, but instead of fixating on any internal layers of BERT, we are fine-tuning all the layers of BERT for only a few epochs before freezing them and then train only the external four Dense layers we have added on top of BERT. The reason for even training the BERT layers is that, the original data on which BERT is trained on is very different from Tweet language. Tweets are very informal and contain a lot more noise. So for the model to learn the semantics of our data, we are training the BERT layers initially.



Figure 12: Training Strategy 1

This strategy actually worked very well compared to previous models. It gave us the best accuracy and F1 score for the 100% of Test data. Since we are training the model for more number of epochs, after freezing the BERT layer, we were able to get much more stable results.

Model	Test Accuracy	Test F1 Score	Kaggle Score
BERT + Training Strategy 2	0.8467	0.8090	0.8517

5.3.4 BERT Embeddings + External Classifiers

In training strategy 2, we were still using Neural fully-connected layers as our classifier. Since BERT generates great Text embeddings, we can use them as the feature vectors for training a different classifier. As part of our experimentation, we implemented two classifiers: SVM & XGBoost. We already used these classifiers but with feature vectors generated by methods such as Bag of words, Count Vectorizer and TF-IDF. Here, we aim to generate the best embeddings possible by BERT. To make this possible, we again train our full model with the Neural classifier for an epoch or two, for the reasons of BERT learning the semantics of our Tweet data. Once this is done, we will instantiate a sub-model that outputs, the output of BERT layers by taking a Encoded tweet as an input. We will use these 1024 dimension vectors as feature vectors for our classifiers.

Model	Test Accuracy	Test F1 Score	Kaggle Score
BERT + SVM	0.8427	0.8079	0.8456
BERT + XGBoost	0.8369	0.7995	0.8435

There's a huge jump in metrics from the initial implementations of the same model. Almost a 5% jump in Kaggle scores. It still didn't beat the Neural classifier metrics, but with SVM it's really close to it.

5.4 Hyperparameter Tuning and Testing

We took a very manual approach to hyperparameter tuning owing to the model run times and availability of computational resources to us. With the Vanilla BERT, we extensively tested a set of parameter and established ranges or sustained lists for each of them. For every BERT based model and approach, we limited our search space to these established ranges and lists without any cross-validation procedures.

As per the Testing, the dataset comes with a completely different set of Testing samples on which we reported all our model evaluation metrics. We initially split our training set into train and validation while tuning the parameters, but once we found the parameters, we went back and retrained the model on the entire Training set without holding out any data. The evaluation metrics on Test data are reported with a model trained in the above mentioned way.

6 Final Results

In this section, we present the results for the best performing model of each of our Model variations. Primarily, we have implemented SVM, XGBoost, Bi-directional LSTM and BERT as a classifier. But again in BERT we have variations based on the final classifier used and the training strategy followed.

Our best Kaggle scores are achieved by Vanilla BERT and puts us in the top 10% of the leaderboard. But, our best model is **BERT + Neural Classifier + Training Strategy 2** which gave the best

Model	Test Accuracy	Test F1 Score	Kaggle Score
SVM	0.7974	0.7271	0.7995
XGBoost	0.7787	0.7303	0.8016
LSTM	0.8139	0.7666	0.8026
Vanilla BERT	0.8461	0.8086	0.8548
BERT + Training Strategy 1	0.8464	0.8082	0.8527
BERT + Training Strategy 2	0.8467	0.8090	0.8517
BERT + SVM	0.8427	0.8079	0.8456
BERT + XGBoost	0.8369	0.7995	0.8435

accuracy and F1 score on the 100% of Test data and also gave more stable results on each run. Unlike Vanilla BERT, which was staggering between 80% to 84% Kaggle scores, our best model was varying between 83% to 85%.

7 Tools

A better project planning means a better project. It saves time and improves internal communication. The whole team can check their own or the project's progress by simply looking at the dashboard. Our team utilized two such tools for effective project management which are Trello and Kaggle Kernels.

7.1 Trello

Our team utilized Trello, an online free tool for task assignment and internal communication. The dashboard comprised of categories like; Doing, Testing, To-Do, Done, Scheduled Meetings, Backlog and Code Reviews. Every task in the dashboard is assigned a group mentioned above along with a deadline. Our team realized the true potential of this tool during the COVID-19 pandemic. When meeting in person is not an option. Please refer 'references' section for our dashboard link. This is the screenshot of our team's dashboard.

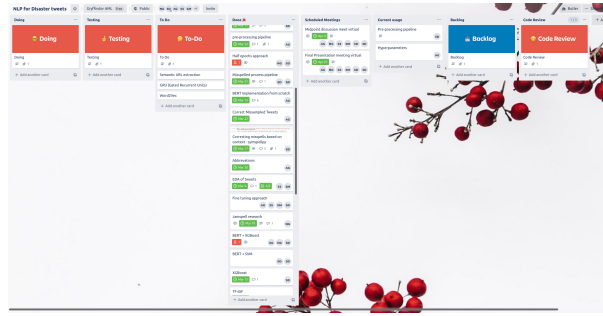


Figure 13: Project Management Tool - Trello

7.2 Kaggle Kernels

Kaggle Kernel is a cloud computational environment that enables reproducible and collaborative analysis. They are essentially Jupyter notebooks in the browser that can be run right before your eyes, all free of charge. Along with the computation on cloud, they also provide features like version control and fork which proved to be very useful for the completion of this project. Our team of 6 was able to collaborate on a same notebook with the help of these kernels.

8 Conclusion

Tweets, as we have been discussing, are very informal and go beyond the usual spoken language. People express their emotions with emojis, they call for attention by using user mentions, they

Submissions and Descriptions	Public Score	Use for Final Score
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input type="checkbox"/>
Test classification with BERT4GROSS (version 12/1)	0.84055	<input checked="" type="checkbox"/>

Figure 14: Version control tool - Kaggle kernels

highlight a specific topic using hashtags. Some overly use punctuation to put more perspective while others use aberrative abbreviations to escape the limitations on the length of the tweet. People write tweets in a way that other humans can find them interesting and informative, not for a second they will think about the ability of the machines to understand the same. The responsibility of giving that ability to machines fall upon us, the fellow data scientists and engineers. While the motivation for our project is powerful, along with the impact that it can create in real-time we are far from making a production-ready application that can be deployed. There are challenges that we will need to address and tasks that we need to streamline better.

As part of this project, we got to experience the fundamentals of Natural Language Processing, the dataset we picked made it only more interesting and challenging to work on this task. We spent considerable amount of time in preprocessing the tweets to make them as readable as possible by the machine. We have tried multiple models and decided to go forward with BERT after seeing its results firsthand and understanding why it is a state-of-the-art NLP system. We may be far from our dream for this project, but we still got great results in terms of accuracy and Kaggle leaderboard position. Our final standings are given in Figure 15.

221	Gryffindor	0.85480	38	6d
Your Best Entry ↑				

Figure 15: Current standing on Kaggle Leaderboard

The best thing we got out of this project is to learning to work together even when we are thousands of miles away, even when our time zones do not match. With the recent developments in the world, we did not have the choice of working from a single location and using tools like Trello really helped us in streamlining our work and made remotely working entirely possible.

9 Future Work

For a very long period, spanning this project, we were stuck at the 85% Kaggle score mark, along with 30 others. This seems like a huge local minima in the loss curve for the test set. Our training strategies helped us in making our training results more stable but did not help in escaping this minima. We can explore more in terms of preprocessing steps or better reinforcements to our model to deal with noisy labelling.

The inference time for our combined preprocessing steps and prediction is not even close to being a realistic deployable model to filter tweets in real-time. More than the predictions, our preprocessing steps are taking over 70% of inference for a sample. We will need to build more efficient and economical methods that are more streamlines to our tasks rather than using out-of-the-box libraries to improve the inference time.

Even though BERT gives us state-of-the-art results, we will need to explore the model architecture for better Fine-tuning. Recent studies also point towards ALBERT working better in text classification tasks compared to BERT.

References

- [1] <https://www.kaggle.com/c/nlp-getting-started>
- [2] <https://towardsml.com/2019/09/17/bert-explained-a-complete-guide-with-theory-and-tutorial/>
- [3] <https://ai.googleblog.com/2018/11/open-sourcing-bert-state-of-art-pre.html>
- [4] https://xgboost.readthedocs.io/en/latest/tutorials/param_tuning.html
- [5] Jacob Devlin, Ming-Wei, Chang Kenton, & Lee Kristina, Toutanova (2019)BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding *arXiv***1810** (04805v2) [cs.CL] 24 May 2019
- [6] <https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/>
- [7] <https://towardsdatascience.com/hyperparameter-optimization-with-keras-b82e6364ca53>
- [8] <https://medium.com/dissecting-bert/dissecting-bert-part-1-d3c3d495cdb3>
- [9] <https://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras/>
- [10] <https://medium.com/@raghavaggarwal0089/bi-lstm-bc3d68da8bd0>
- [11] <https://pypi.org/project/sympellpy/>