CLASSIFICATION OF DISASTER TWEETS

UDACITY – MACHINE LEARNING NANODEGREE

CAPSTONE PROJECT

Bikram Dutta
Mumbai
https://github.com/bikramdutta
https:// www.linkedin.com/in/bikramdutta

ABSTRACT

In this project, our objective is to create a machine learning model which is capable of classifying a tweet into either a tweet related to a disaster or not. We analyse, clean and process the tweets (natural language) and build models to serve our purpose. As the problem is part of a Kaggle Competition, data is provided by Kaggle. Exploratory Data Analysis is carried out and important inferences are drawn to further drive our problem solving. The state of art NLP models such as BERT and TF-IDF are used. The results are compared for a number of variations in models and their hyperparameters. Tuning is done accordingly to come to the best results possible. In the end the best results are submitted as a part of Kaggle Competition Submission.

1. Definition

1.1 Project Overview

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. Because of this, more agencies are interested in programmatically monitoring Twitter in this case, the disaster relief organizations and news agencies. But, it's not always clear whether a person's words are actually announcing a disaster.
For e.g.: "On plus side LOOK AT THE SKY LAST NIGHT IT WAS ABLAZE"
The author explicitly uses the word "ABLAZE" but means it metaphorically. This is clear to a human right away, especially with the visual aid. But it's less clear to a machine.
In this competition, I am challenged to build a machine learning model that predicts which Tweets are about real disasters and which one's aren't. I have access to a dataset of 10,000 tweets that were hand classified.[1]

The data has been provided by Kaggle for this particular competition: -
https://www.kaggle.com/c/nlp-getting-started/data
But the dataset was originally created by the company figure-eight and originally shared on their 'Data For Everyone' website here.Tweet source: https://twitter.com/AnyOtherAnnaK/status/629195955506708480[1]

3 files are provided: -
1. train.csv
2. test.csv
3. sample_submission.csv

1.2 Problem Statement

The challenge is to build a machine learning model that predicts which Tweets are about real disasters and which one's aren't.

A set of 10,000 tweets is given. For a subset i.e. tweets in train.csv, the classification is already given. For the other subset, i.e. tweets in test.csv, the classification is to be done. [1]

The data is human written text, natural language data which has to be processed, and visualized for inferences which will would help create the model. [1]

1.3 Metrics

It is a binary classification problem and therefore, the accuracy and F1 score are of much significance. We also consider other metrics. In a nutshell there are 4 metrics involved: -

- Accuracy measures the fraction of the total sample that is correctly identified
- Precision measures that out of all the examples predicted as positive, how many are actually positive
- Recall measures that out of all the actual positives, how many examples were correctly classified as positive by the model
- F1 Score is the harmonic mean of the Precision and Recall

Classification Report of sklearn.metrics.classification_report, computes those metrics for the given training and validation set. [6]

2. Analysis

2.1 Data exploration

2.1.1 Missing Values

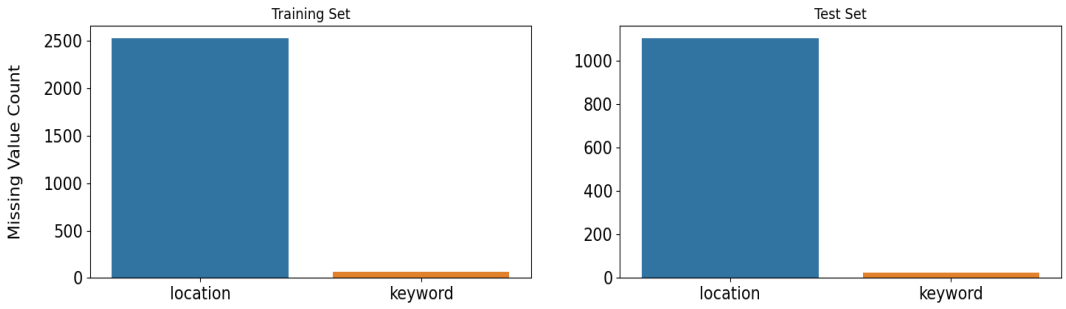
The training data and the test data are scanned for missing values.

Training set

Fields	**Total Missing Values**
location	2533
keyword	61
target	0
text	0
id	0

Test set

Fields	**Total Missing Values**
location	1105
keyword	26
text	0
id	0



Inferences: -

- Both training and test set have same ratio of missing values in keyword and location.
- 0.8% of keyword is missing in both training and test set
- 33% of location is missing in both training and test set
- Since missing value ratios between training and test set are too close, they are most probably taken from the same sample. Missing values in those features are filled with no_keyword and no_location respectively.

2.1.2 Cardinality

- Number of unique keywords: 222 in training set, and 222 in test set.
- Number of unique locations: 3342 in training set, and 1603 in test set.

Inferences: -

There are many unique values in the field – location. It shouldn't be used as a feature. This is probably because it is a user input and there are no set values to be chosen or they are not configured to be automatically generated.

2.1.3 Ambiguous Samples

There are many tweets which have appeared more than once in the dataset. That is fine. Many people c an be tweeting the exact same string and it can appear in our dataset. The only thing is the classification of that tweet should beuniform. So, the tweet should belong to only one class at a time.

We once check whether there is any tweet which has appeared in both the classes

text	No. of Unique Target Value
like for the music video I want some real action shit like burning buildings and police chases not some weak ben winston shit	2
Hellfire! We don‰Ûªt even want to think about it or mention it so let‰Ûªs not do anything that leads to it #islam!	2
The Prophet (peace be upon him) said 'Save yourself from Hellfire even if it is by giving half a date in charity.'	2
In #islam saving a person is equal in reward to saving all humans! Islam is the opposite of terrorism!	2
To fight bioterrorism sir.	2
Who is bringing the tornadoes and floods. Who is bringing the climate change. God is after America He is plaguing her\n \n#FARRAKHAN #QUOTE	2
#foodscare #offers2go #NestleIndia slips into loss after #Magginoodle #ban unsafe and hazardous for #humanconsumption	2
#Allah describes piling up #wealth thinking it would last #forever as the description of the people of #Hellfire in Surah Humaza. #Reflect	2
He came to a land which was engulfed in tribal war and turned it into a land of peace i.e. Madinah. #ProphetMuhammad #islam	2
RT NotExplained: The only known image of infamous hijacker D.B. Cooper. http://t.co/JlzK2HdeTG	2
Hellfire is surrounded by desires so be careful and don‰Ûªt let your desires control you! #Afterlife	2
CLEARED:incident with injury:I-495 inner loop Exit 31 - MD 97/Georgia Ave Silver Spring	2
Mmmmmm I'm burning.... I'm burning buildings I'm building.... Ooooooohhhh oooh ooh...	2
wowo--=== 12000 Nigerian refugees repatriated from Cameroon	2
.POTUS #StrategicPatience is a strategy for #Genocide; refugees; IDP Internally displaced people; horror; etc. https://t.co/rqWuoy1fm4	2
Caution: breathing may be hazardous to your health.	2
I Pledge Allegiance To The P.O.P.E. And The Burning Buildings of Epic City. ??????	2
that horrible sinking feeling when you‰Ûªve been at home on your phone for a while and you realise its been on 3G this whole time	2

It is observed that there are **18** unique tweets in training set which are classified differently in their duplicates.

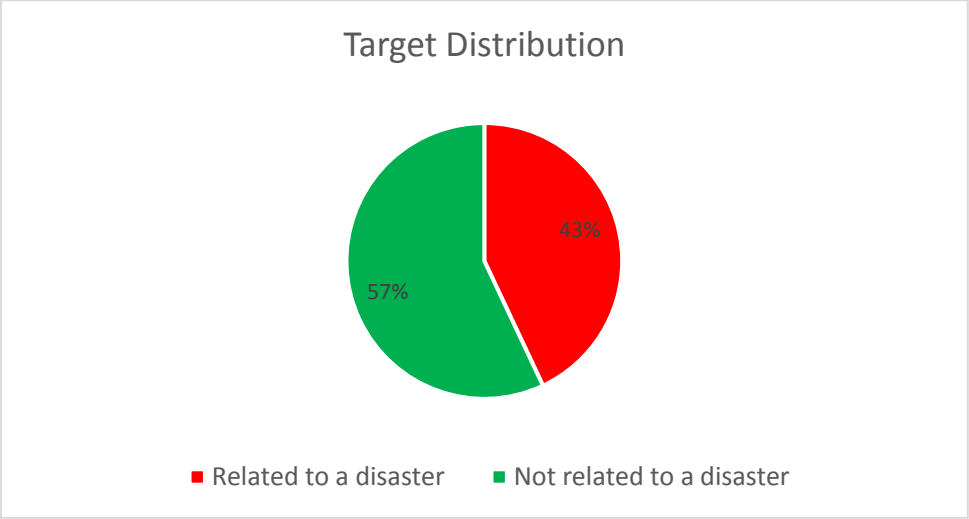
The classification has been done manually by humans and it is possible that those tweets are probably labelled by different people and they interpreted the meaning differently because some of them are not very clear. Therefore, tweets with two unique target values have to **be** relabelled (reclassified) since they can affect the training score.

2.2 Exploratory Visualization

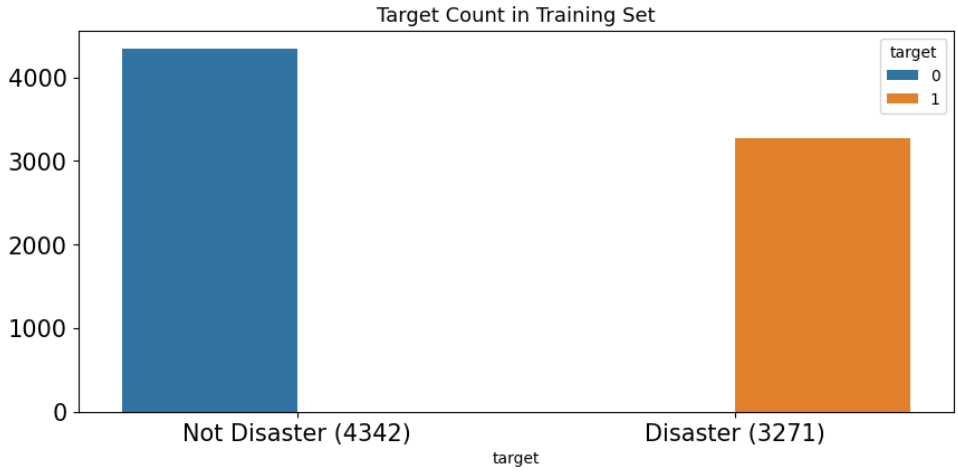
2.2.1 Target Distribution

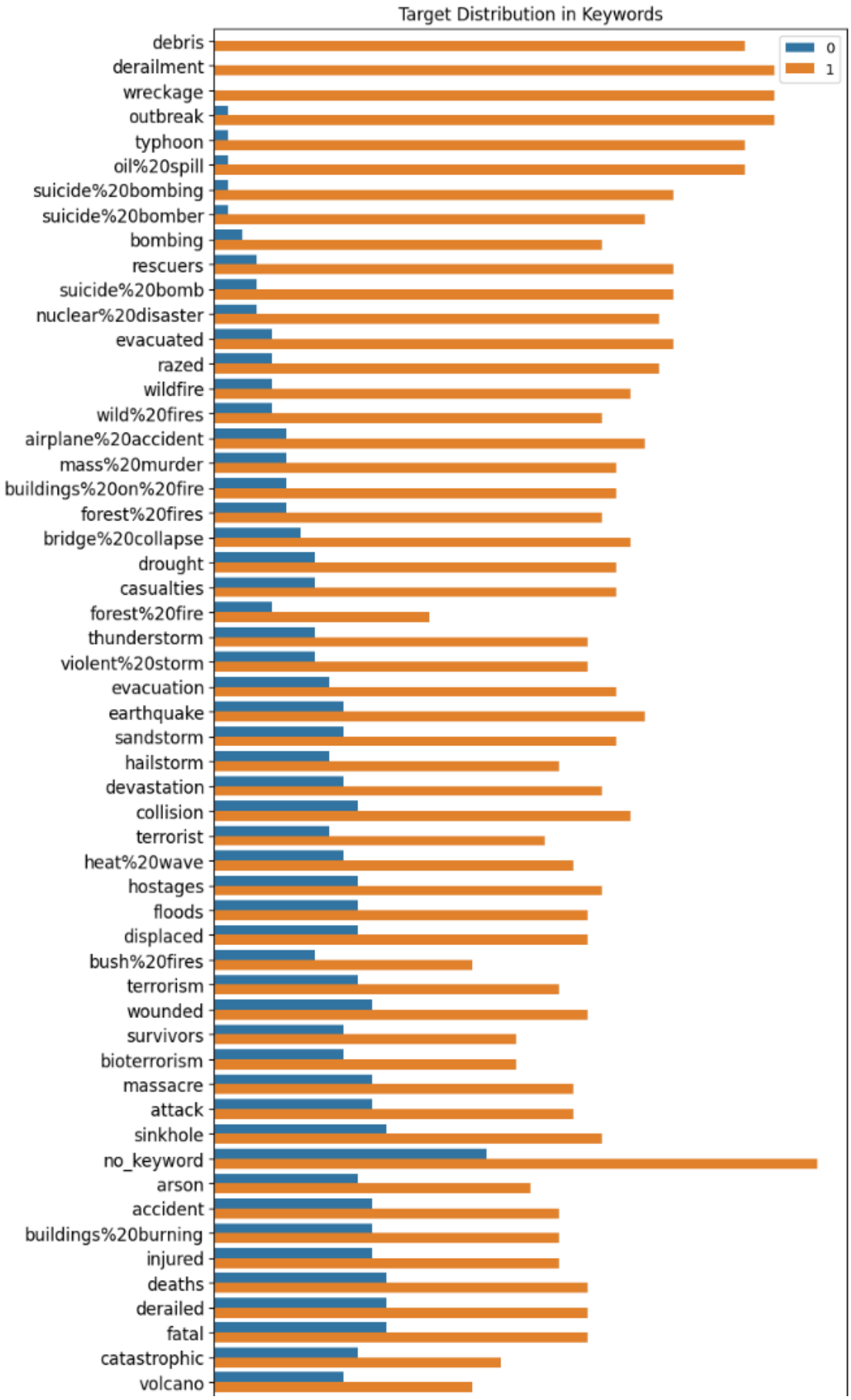
Fortunately, there is signal in keyword because some of those words can only be used in one context. Keywords have very different tweet counts and target means. keyword can be used as a feature by itself or as a word added to the text. Every single keyword in training set exists in test set. If training and test set are from the same sample, it is also possible to use target encoding on keyword.

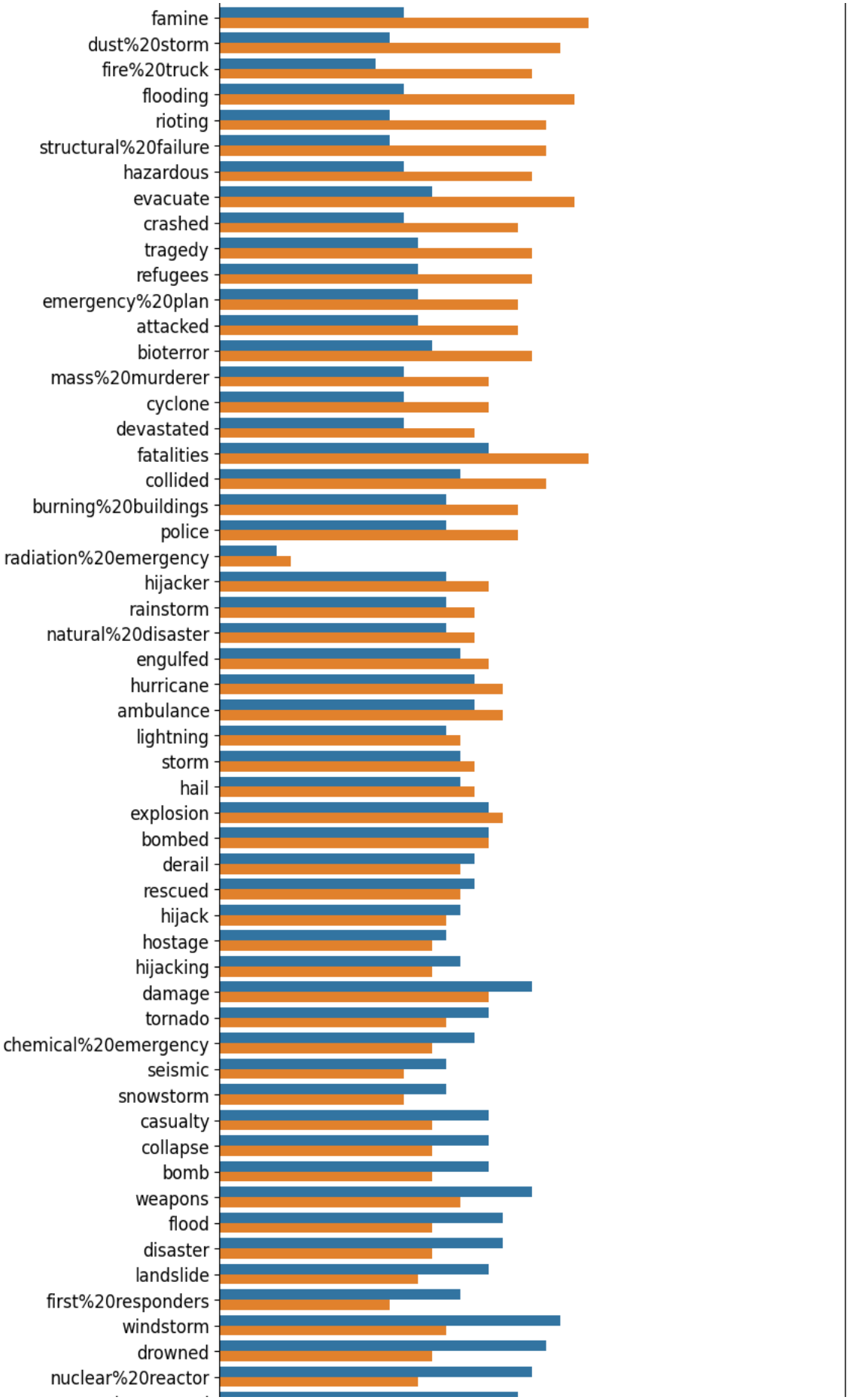
The following pie chart shows the distribution of disaster and non-disaster tweets: -

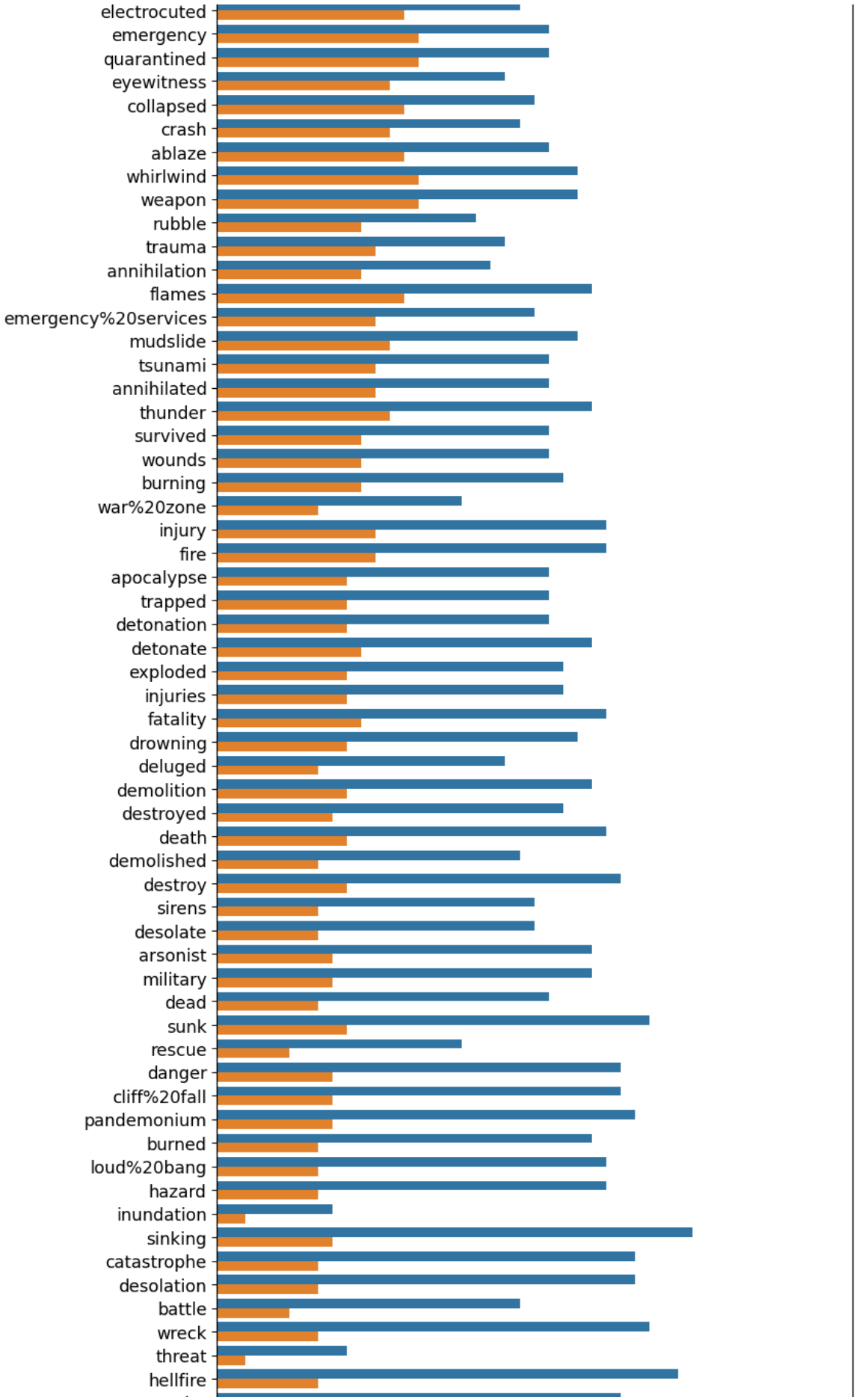


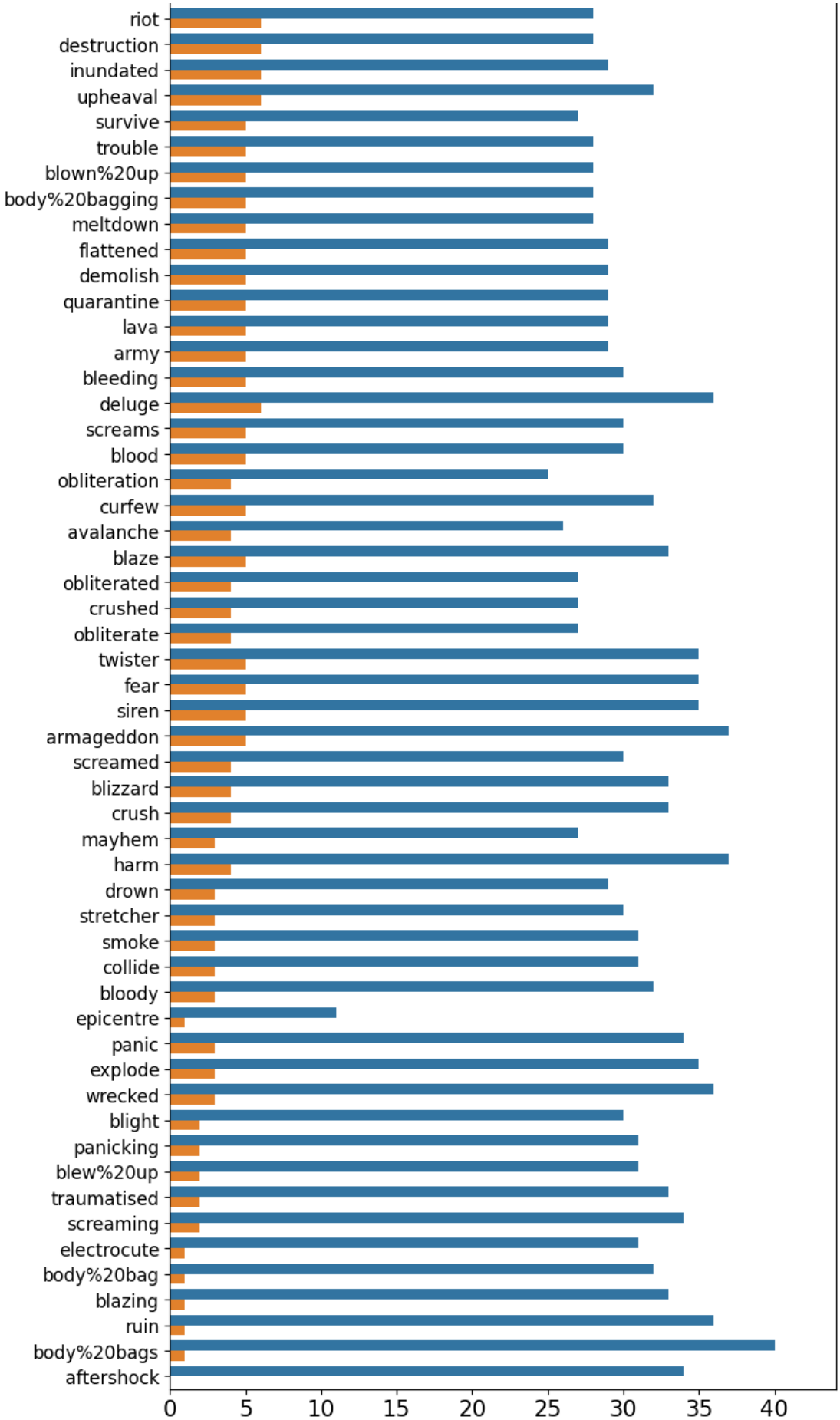
The following bar chart shows the counts of disaster and non disaster tweets in the data: -




Target Distribution in Keywords
0
1
debris
derailment
wreckage
outbreak
typhoon
oil%20spill
suicide%20bombing
suicide%20bomber
bombing
rescuers
suicide%20bomb
nuclear%20disaster
evacuated
razed
wildfire
wild%20fires
airplane%20accident
mass%20murder
buildings%20on%20fire
forest%20fires
bridge%20collapse
drought
casualties
forest%20fire
thunderstorm
violent%20storm
evacuation
earthquake
sandstorm
hailstorm
devastation
collision
terrorist
heat%20wave
hostages
floods
displaced
bush%20fires
terrorism
wounded
survivors
bioterrorism
massacre
attack
sinkhole
no_keyword
arson
accident
buildings%20burning
injured
deaths
derailed
fatal
catastrophic
volcano


famine
dust%20storm
fire%20truck
flooding
rioting
structural%20failure
hazardous
evacuate
crashed
tragedy
refugees
emergency%20plan
attacked
bioterror
mass%20murderer
cyclone
devastated
fatalities
collided
burning%20buildings
police
radiation%20emergency
hijacker
rainstorm
natural%20disaster
engulfed
hurricane
ambulance
lightning
storm
hail
explosion
bombed
derail
rescued
hijack
hostage
hijacking
damage
tornado
chemical%20emergency
seismic
snowstorm
casualty
collapse
bomb
weapons
flood
disaster
landslide
first%20responders
windstorm
drowned
nuclear%20reactor


electrocuted
emergency
quarantined
eyewitness
collapsed
crash
ablaze
whirlwind
weapon
rubble
trauma
annihilation
flames
emergency%20services
mudslide
tsunami
annihilated
thunder
survived
wounds
burning
war%20zone
injury
fire
apocalypse
trapped
detonation
detonate
exploded
injuries
fatality
drowning
deluged
demolition
destroyed
death
demolished
destroy
sirens
desolate
arsonist
military
dead
sunk
rescue
danger
cliff%20fall
pandemonium
burned
loud%20bang
hazard
inundation
sinking
catastrophe
desolation
battle
wreck
threat
hellfire

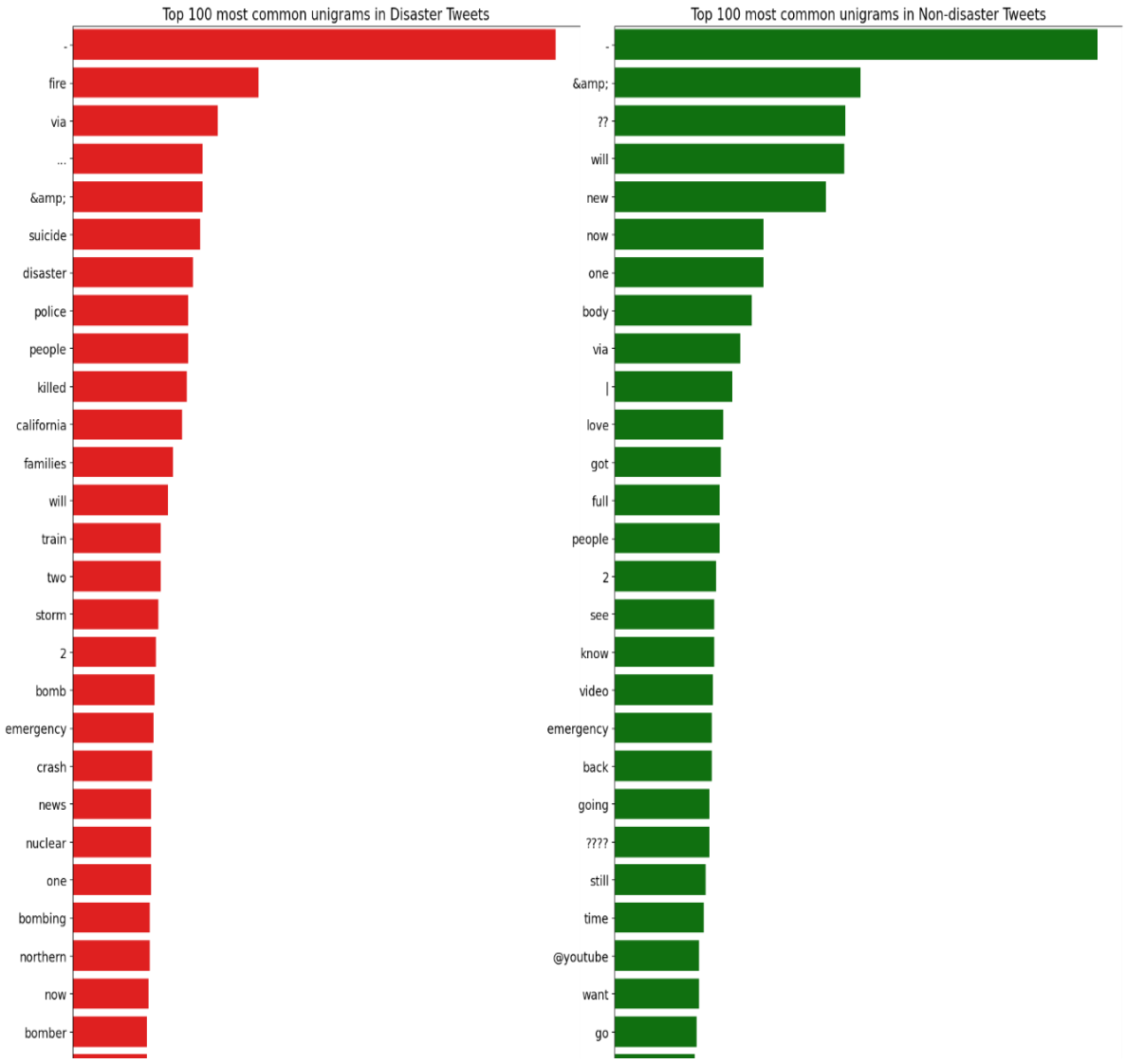

riot
destruction
inundated
upheaval
survive
trouble
blown%20up
body%20bagging
meltdown
flattened
demolish
quarantine
lava
army
bleeding
deluge
screams
blood
obliteration
curfew
avalanche
blaze
obliterated
crushed
obliterate
twister
fear
siren
armageddon
screamed
blizzard
crush
mayhem
harm
drown
stretcher
smoke
collide
bloody
epicentre
panic
explode
wrecked
blight
panicking
blew%20up
traumatised
screaming
electrocute
body%20bag
blazing
ruin
body%20bags
aftershock
0
5
10
15
20
25
30
35
40

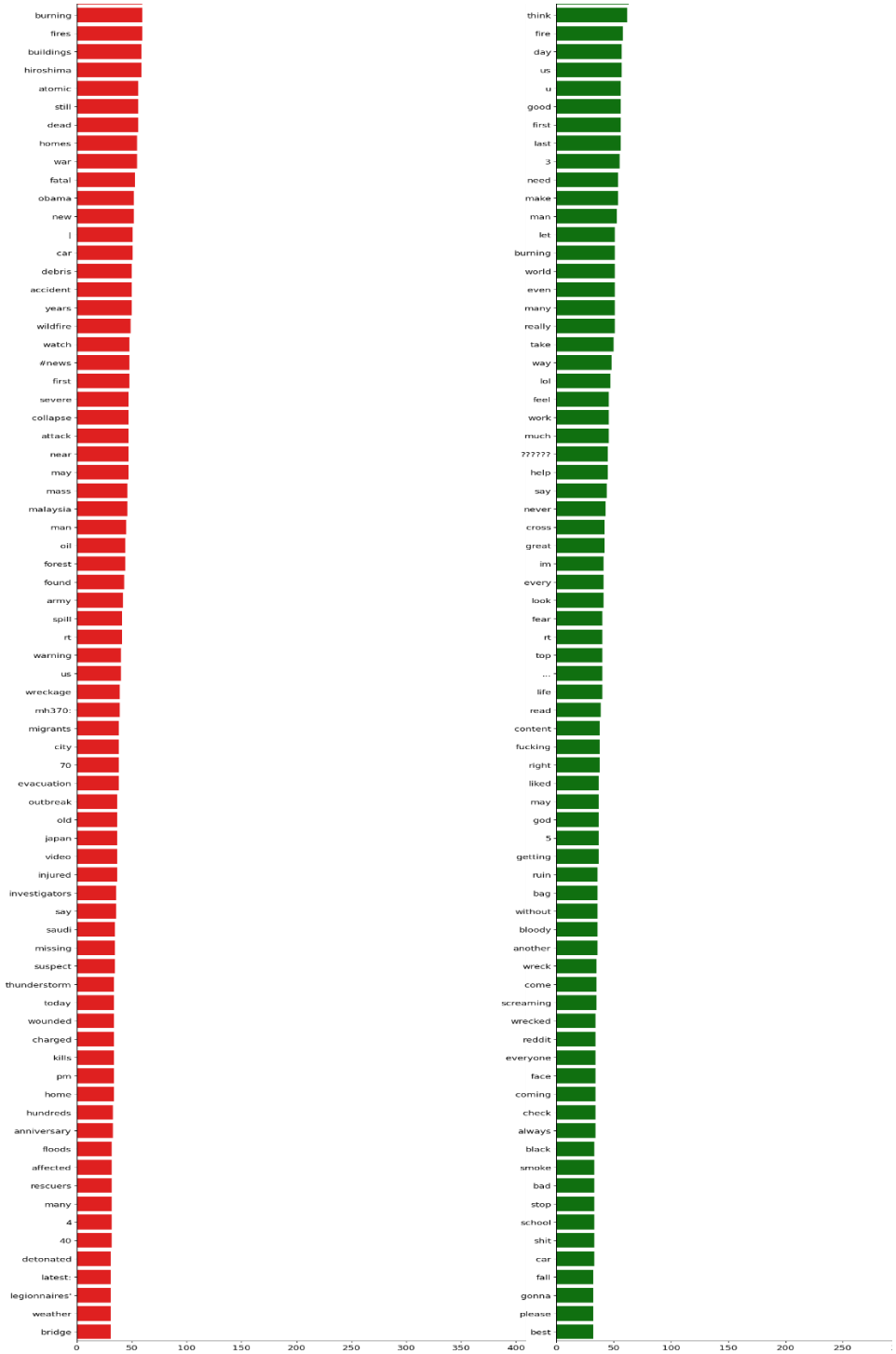
2.2.2 Text mining: N-grams

Text mining, also known as text analysis, is the process of transforming unstructured text data into meaningful and actionable information.

N-grams of texts are extensively used in text mining and natural language processing tasks. They are basically a set of co-occurring words within a given window and when computing the n-grams you typically move one word forward (although you can move X words forward in more advanced scenarios). We extracted the n-grams and observed the first 100 n-grams (unigrams, bigrams and trigrams). We deduced the following inferences based on that. [2][3]

UNIGRAMS (1-grams)





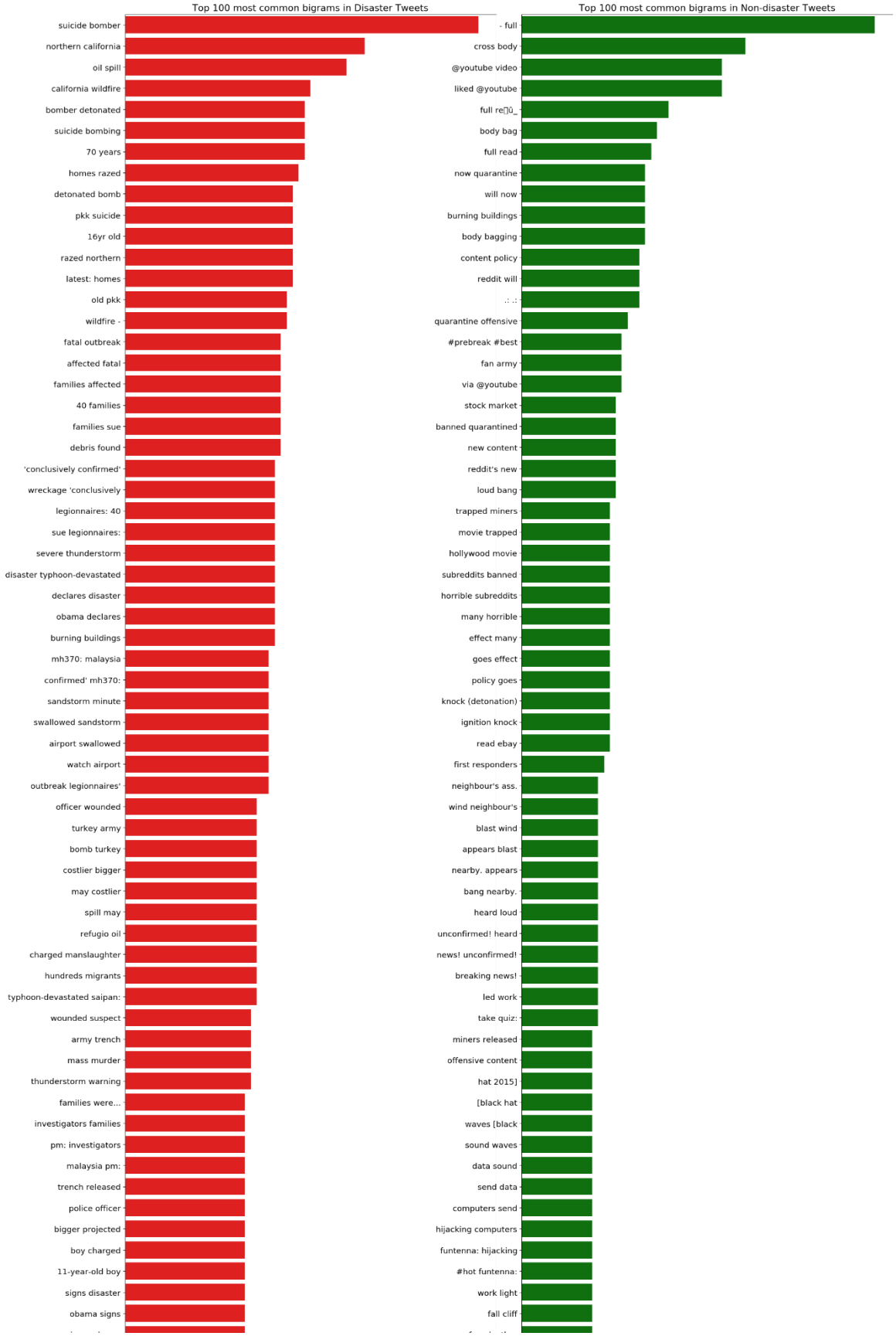
Inferences: -

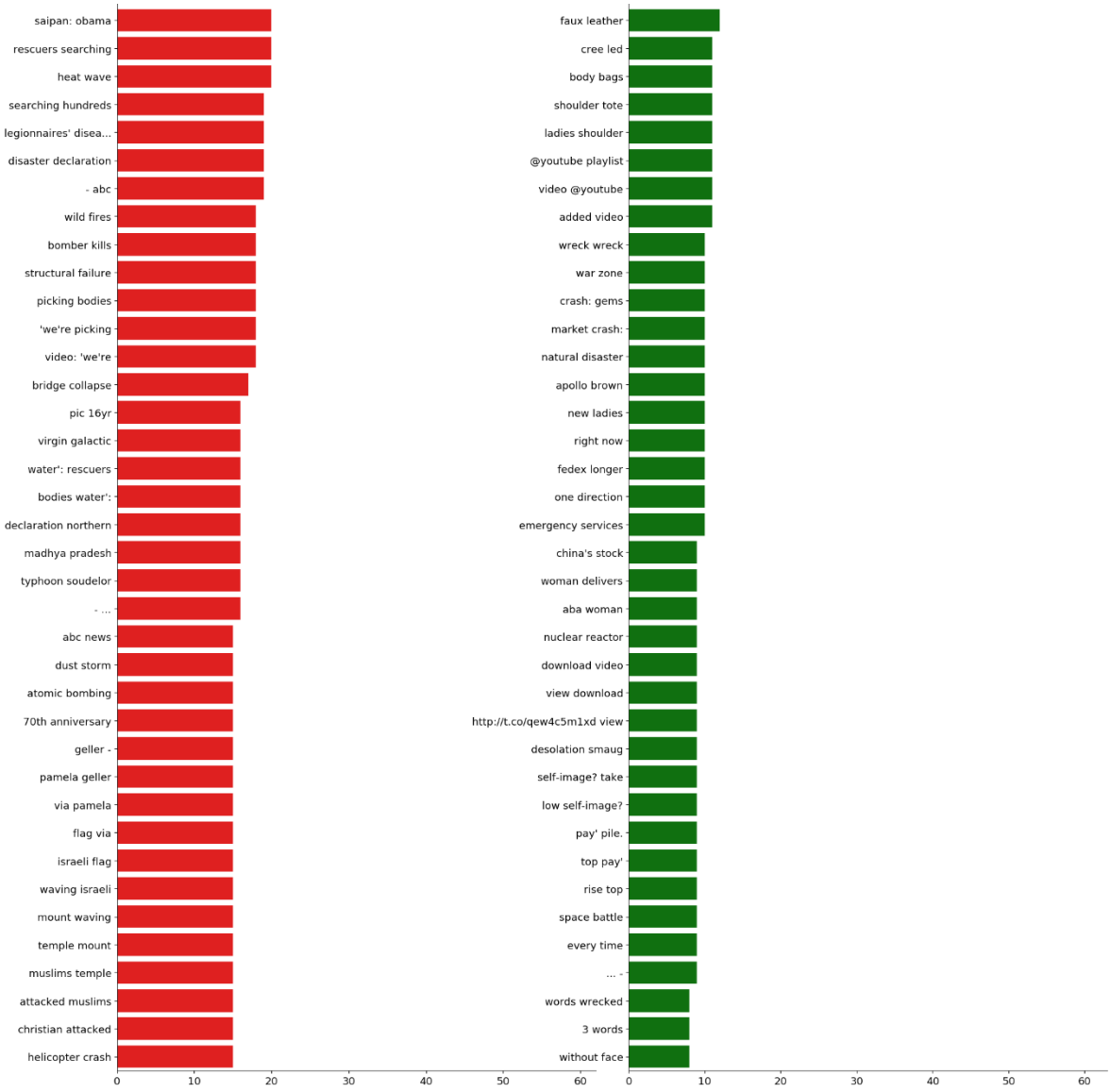
It is observed that the most frequent unigrams exist in both the classes (0/1). Mostly, the unigrams are numbers, punctuations or stop words. These elements don't provide any distinguishing information for the model to work, therefore it is required to eradicate these elements and clean the data.

Also, the most frequent unigrams in disaster tweets are already giving information about disasters. It is difficult to use some of those words in other contexts.

And, most frequent unigrams in non-disaster tweets are verbs. This is because we are dealing with tweets and in most of the cases it has informal language and active voice.

BIGRAMS (2-grams)





Inferences: -

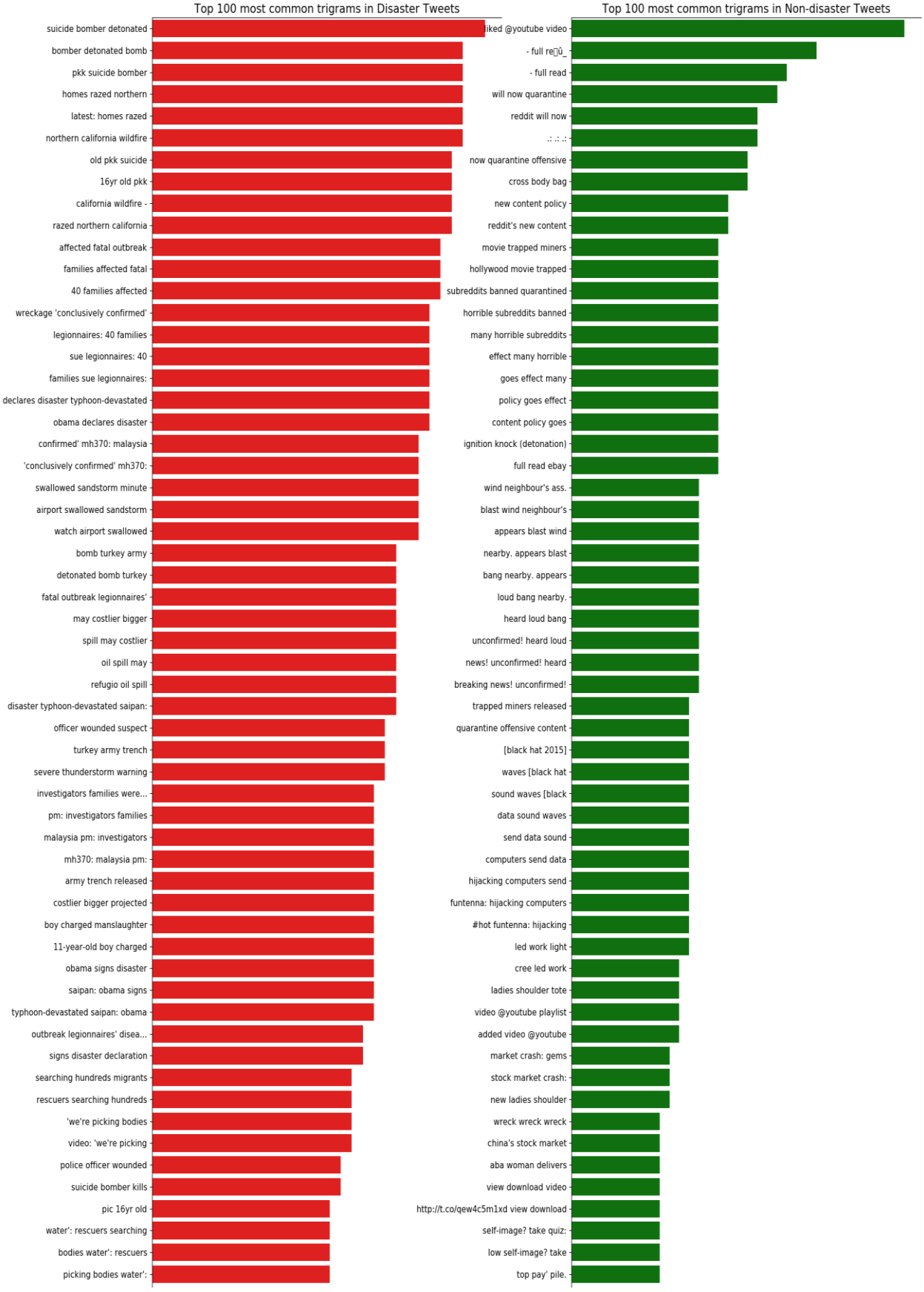
It is observed that there are no common bigrams in both the classes (0/1). It means the context is clear.

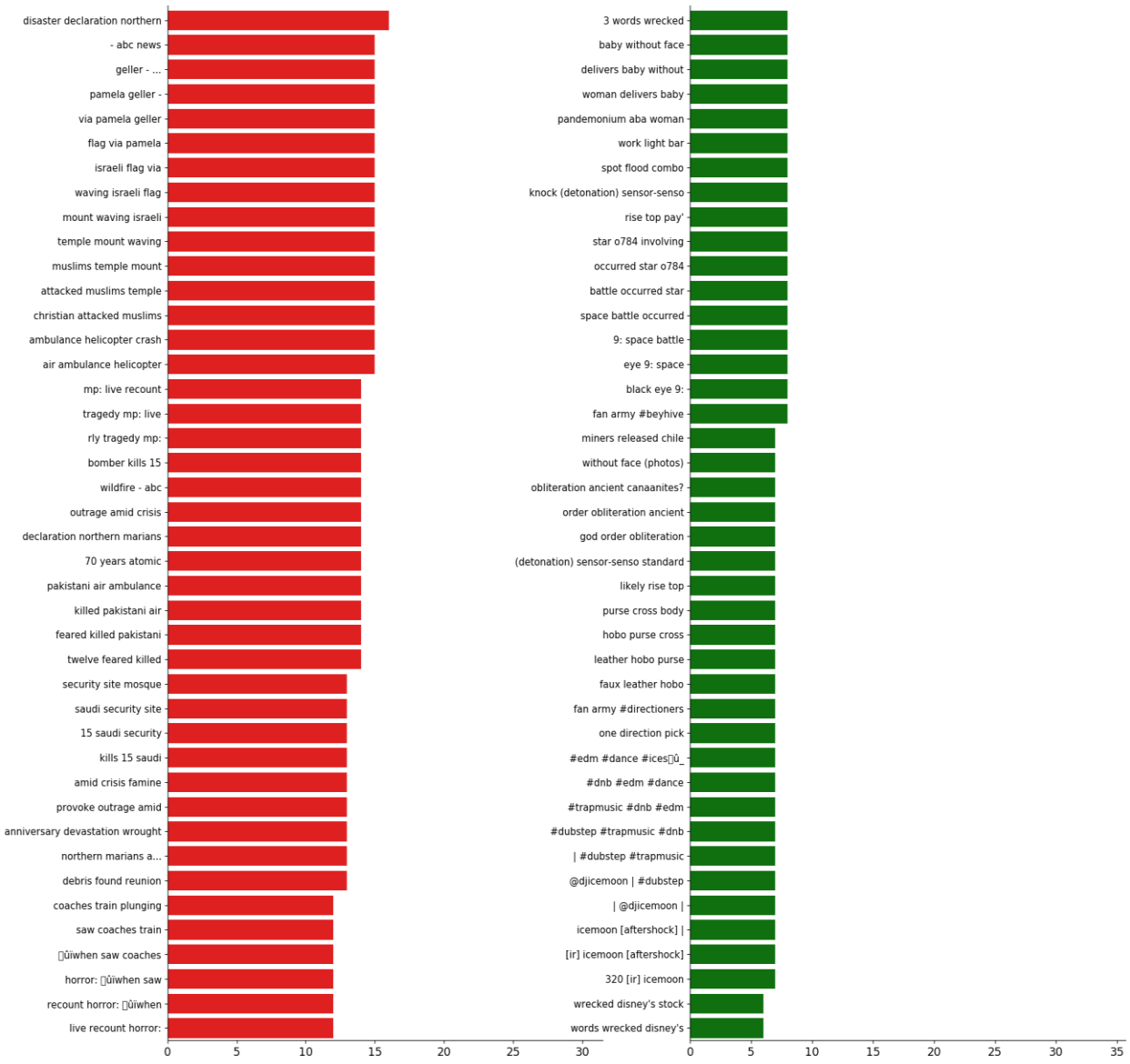
Most frequent bigrams in disaster tweets give out more information about the disaster than what we have observed in the case of unigrams.

And, for the non-disaster tweets, the most frequent bigrams are related to YouTube or reddit. Thus, they contain a lot of punctuations.

It is observed that punctuations in many bigrams, which duplicate the form of words and as previously stated don't add distinguishing information for the model. Therefore, it is required to clean the bigrams of all punctuations.

TRIGRAMS (3-grams)





Inferences: -

As observed in the case of bigrams, there are no common trigrams in both the classes (0/1) and thereby the context is clearer.

For the disaster tweets, the trigrams give out information on the disaster like in the case of bigrams.

For the non-disaster tweets, the case is the same as the bigrams and hence a lot of involvements of punctuations. And therefore, it is required to clean the trigrams also of all punctuations.

2.2.3 Meta features

Tweets are not simple string of texts, but from which different types of inferences could be drawn. Also, tweets have features like mentions, hashtags (for trends), urls, etc. Taking those data along with text stats, into consideration would be really useful to judge the data. Distributions of meta features in classes and datasets can be helpful to identify disaster tweets. [6]

I extract the following meta features from the data: -

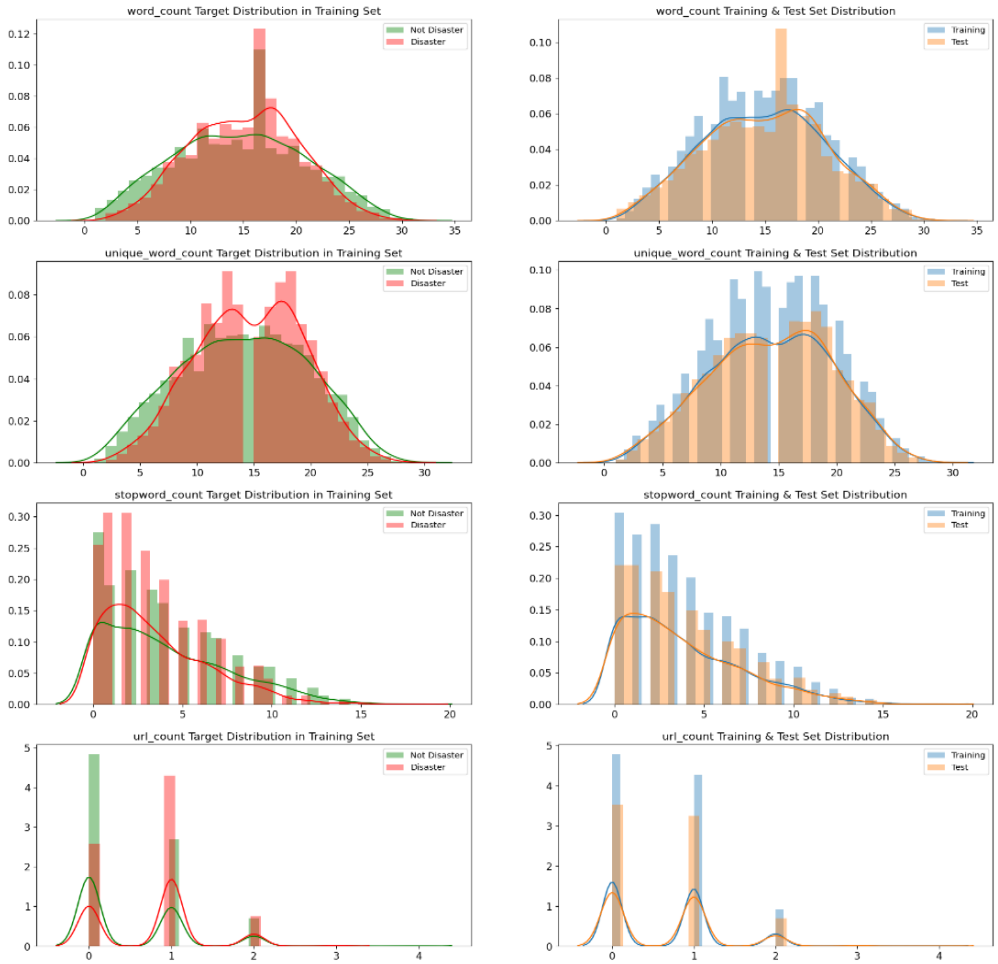
- word_count number of words in text
- unique_word_count number of unique words in text
- stop_word_count number of stop words in text
- url_count number of urls in text
- mean_word_length average character count in words
- char_count number of characters in text
- punctuation_count number of punctuations in text
- hashtag_count number of hashtags (#) in text
- mention_count number of mentions (@) in text

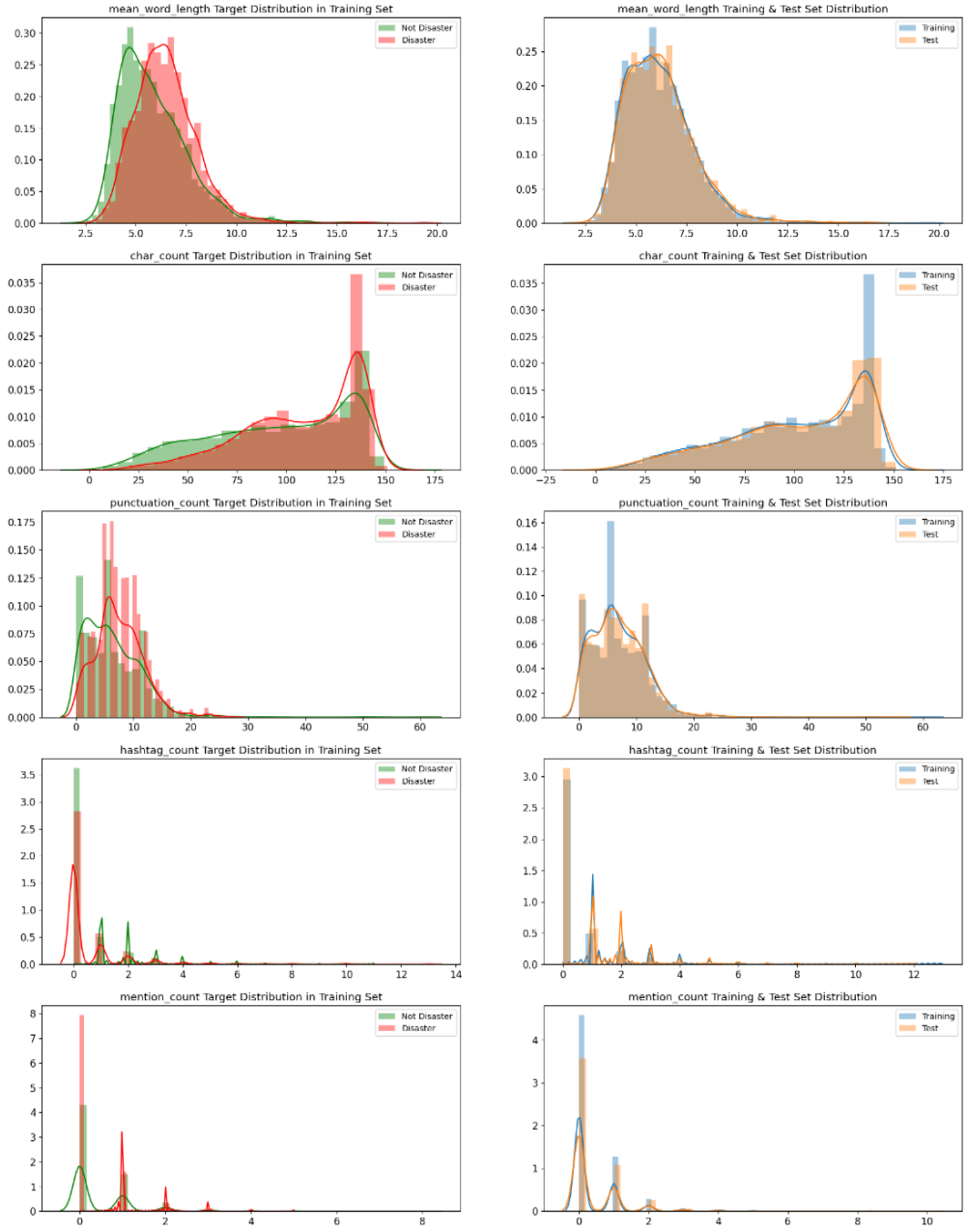
The following operations are involved to extract the meta features: -

```
METAFEATURES = []
# word count
train_df['word_count'] = train_df['text'].apply(lambda tweet: len(str(tweet).split()))
test_df['word_count'] = test_df['text'].apply(lambda tweet: len(str(tweet).split()))
METAFEATURES.append('word_count')
# unique_word_count
train_df['unique_word_count'] = train_df.text.apply(lambda tweet: len(set(str(tweet).split())))
test_df['unique_word_count'] = test_df.text.apply(lambda tweet: len(set(str(tweet).split())))
METAFEATURES.append('unique_word_count')
# stop_word_count
train_df['stopword_count'] = train_df.text.apply(lambda tweet: len([word for word in str(tweet).sp
lit() if word in STOPWORDS]))
test_df['stopword_count'] = test_df.text.apply(lambda tweet: len([word for word in str(tweet).spli
t() if word in STOPWORDS]))
METAFEATURES.append('stopword_count')
# url_count
train_df['url_count'] = train_df.text.apply(lambda tweet: len([word for word in tweet.split() if (
'https' in word or 'http' in word)]))
test_df['url_count'] = test_df.text.apply(lambda tweet: len([word for word in tweet.split() if ('h
ttps' in word or 'http' in word)]))
METAFEATURES.append('url_count')
#mean_word_length
train_df['mean_word_length'] = train_df.text.apply(lambda tweet: np.mean([len(word) for word in tw
eet.split()]))
test_df['mean_word_length'] = test_df.text.apply(lambda tweet: np.mean([len(word) for word in twee
t.split()]))
METAFEATURES.append('mean_word_length')
# char count
train_df['char_count'] = train_df.text.apply(lambda tweet: len(str(tweet)))
test_df['char_count'] = test_df.text.apply(lambda tweet: len(str(tweet)))
METAFEATURES.append('char_count')
```

```
# punctuation_count
train_df['punctuation_count'] = train_df['text'].apply(lambda tweet: len([word for word in str(twe
et) if word in string.punctuation]))
test_df['punctuation_count'] = test_df['text'].apply(lambda tweet: len([word for word in str(tweet
) if word in string.punctuation]))
METAFEATURES.append('punctuation_count')
# hashtag_count
train_df['hashtag_count'] = train_df['text'].apply(lambda tweet: len([word for word in str(tweet)
if word == '#']))
test_df['hashtag_count'] = test_df['text'].apply(lambda tweet: len([word for word in str(tweet) if
 word == '#']))
METAFEATURES.append('hashtag_count')
# mention_count
train_df['mention_count'] = train_df['text'].apply(lambda tweet: len([word for word in str(tweet)
if word == '@']))
test_df['mention_count'] = test_df['text'].apply(lambda tweet: len([word for word in str(tweet) if
 word == '@']))
METAFEATURES.append('mention_count')
```

All of the meta features have information about target, but some of them are not good enough such as url_count, hashtag_count and mention_count. On the other hand, word_count, unique_word_count, stop_word_count, mean_word_length, char_count, punctuation_count have very different distributions for disaster and non-disaster tweets. Those features might be useful in models.





Inferences: -

It looks like the disaster tweets are more scrutinized or are filtered or it is tweeted from news agencies, as they are written in a formal way with lesser short form of words.

For the non-disaster tweets, they contain short form of words and typos, and it can be assumed that they are tweeted from individual users as there are no filters and scrutiny of text observed.

2.3 Algorithm and Techniques

BERT Model

BERT is a deep learning model that has given state-of-the-art results on a wide variety of natural language processing tasks. It stands for Bidirectional Encoder Representations for Transformers. [7] BERT is pre-trained on a large corpus of unlabelled text including the entire Wikipedia (that's 2,500 million words!) and Book Corpus (800 million words). This **pre-training** step is half the magic behind BERT's success. This is because as we train a model on a large text corpus, our model starts to pick up the deeper and intimate understandings of how the language works. This knowledge is the **swiss army knife** that is useful for almost any NLP task. [4] [8]

BERT is a **"deeply bidirectional"** model. Bidirectional means that BERT learns information from both the left and the right side of a token's context during the training phase.

The bidirectionality of a model is important for truly understanding the meaning of a language. Let's see an example to illustrate this. There are two sentences in this example and both of them involve the word "bank":

i. We went to the river bank.
ii. I need to go to the bank to make a deposit.

If we try to predict the nature of the word "bank" by only taking either the left or the right context, then we will be making an error in at least one of the two given examples.

One way to deal with this is to consider both the left and the right context before making a prediction. That's exactly what BERT does! [4] [8] [9]

These embeddings were used to train models on downstream NLP tasks and make better predictions. This could be done even with less task-specific data by utilizing the additional information from the embeddings itself. We will use BERT to extract embeddings from each tweet in the dataset and then use these embeddings to train a text classification model.

This model uses the implementation of BERT from the TensorFlow Models repository on GitHub at tensorflow/models/official/nlp/bert. It uses L=12 hidden layers (Transformer blocks), a hidden size of H=768, and A=12 attention heads. [6]

2.4 Benchmark Model

TF-IDF

TFIDF stands for Term Frequency – Inverse Data Frequency. We use it to score the relative importance of words

From a BoW (Bag of Words) approach all words are broken into count and frequency with no preference to a word in particular, all words have same frequency here and obviously there is no emphasis on important words like collapse, ablaze, etc. by the machine. [5]

Many words which are repeated again and again are given more importance in final feature building and we miss out on context of less repeated but important words like collapse, ablaze, catastrophe, etc.

TF-IDF is useful in solving the major drawbacks of BoW by introducing an important concept called **inverse document frequency.** It's a score which the machine keeps where it is evaluates the words used in a sentence and measures its usage compared to words used in the entire document. In other words, it's a score to highlight each word's relevance in the entire document (set of tweets).

It's calculated as -

Term Frequency (TF)

The number of times a word appears in a document divided by the total number of words in the document. Every document has its own term frequency.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

Inverse Data Frequency (IDF)

The log of the number of documents divided by the number of documents that contain the word ***w***. Inverse data frequency determines the weight of rare words across all documents in the corpus.

$$idf(w) = log(\frac{N}{df_t})$$

TF-IDF

The TF-IDF is simply the TF multiplied by IDF.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

In a nutshell, TF answers the questions like – how many times is 'collapse' use in the entire document (single tweet). And IDF answers the questions like how important is the word – 'collapse' in the entire list of documents (set of tweets) i.e. Is it a frequent theme in most of the documents (tweets)? [5]

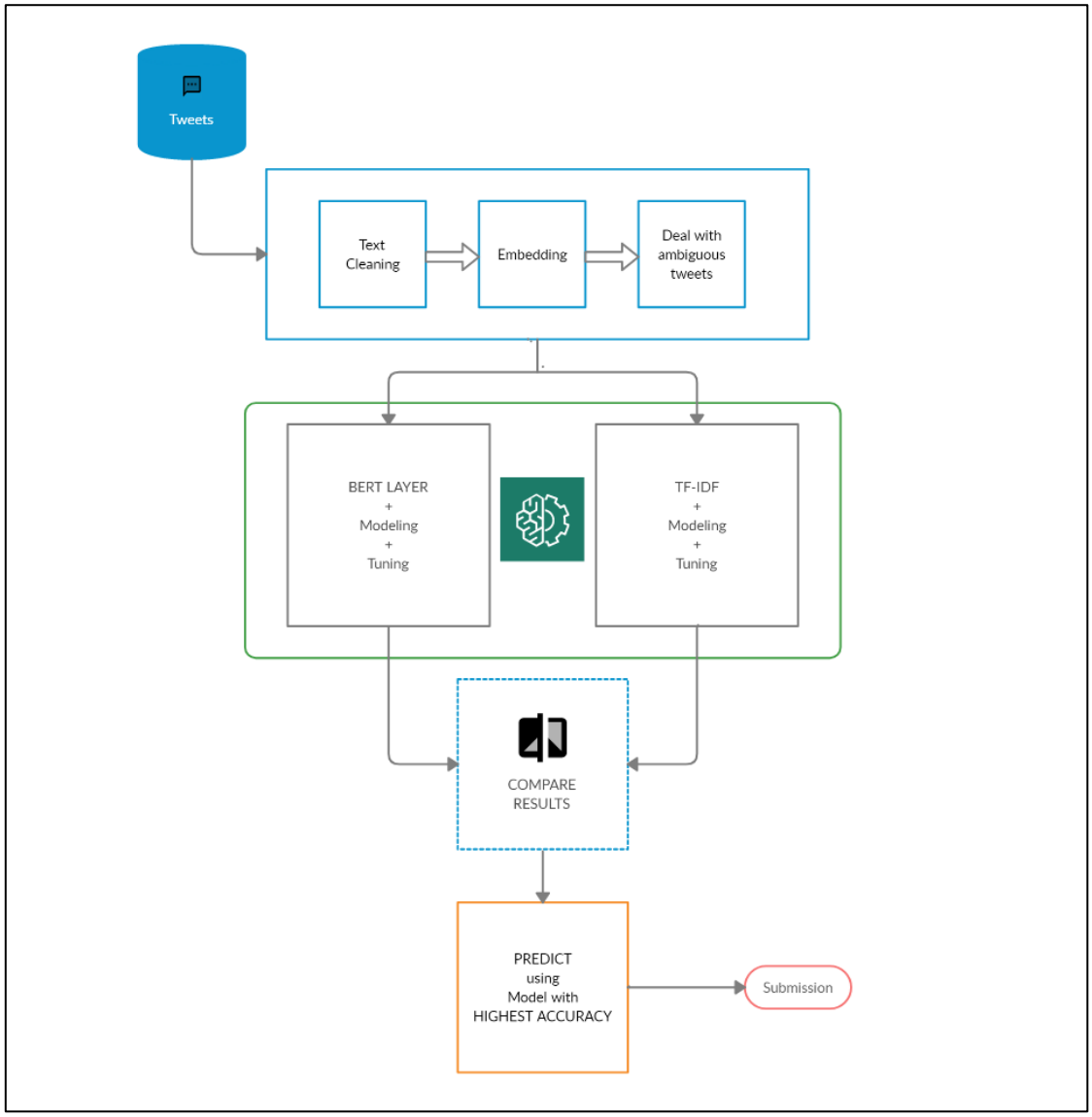
So, using TF and IDF machine makes sense of important words in a document and important words throughout all documents.

We use TFIDF Vectorizer provided by sklearn.feature_extraction.text.TfidfVectorizer[5] to get the vector on which the classification models are to be trained.

The models are trained and tuned to get the best achievable results. The best of the lot is selected as the result.

3. Methodology

The following diagram gives an overview of the process flow: -



3.1 Data pre-processing

3.1.1 Embeddings

Word embeddings are a type of word representation that allows words with similar meaning to have a similar representation. [11]

Simply put, the aim is to extract information on the relationships and correlations between group of words. Multiple different words can go into the same context. For e.g. I painted a bench _____. The blank can be filled with green/red/blue…. Any other colour. So, here the context is same, it can attach the meaning of the blank to that of a colour and thus similar entities can be grouped… Concludingly, blue/red/green can be grouped as they can be used in similar context.

You feed enough examples to a word embedding algorithm for it to figure out the contextual relationships between words; in other words, what words occur near other words.

Word embeddings are in fact a class of techniques where individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped to one vector and the vector values are learned

in a way that resembles a neural network, and hence the technique is often lumped into the field of deep learning.

Key to the approach is the idea of using a dense distributed representation for each word. Each word is represented by a real-valued vector, often tens or hundreds of dimensions. This is contrasted to the thousands or millions of dimensions required for sparse word representations, such as a one-hot encoding.

We can either choose to learn an embedding by feeding a large number of words i.e. feeding all the words in the data set and extract the contextual relations and thus group words or we can choose to go ahead with a pretrained embedding.

For this particular project, we resort to use 2 pretrained embeddings, which are available for free to download.

When we have pre-trained embeddings, doing standard pre-processing steps might not be a good idea because some of the valuable information can be lost. It is better to get vocabulary as close to embeddings as possible. In order to do that, train vocab and test vocab are created by counting the words in tweets.[6]

Text cleaning is based on the embeddings below:-

- GloVe-300d-840B
- FastText-Crawl-300d-2M

```
%%time
glove_embeddings = np.load('/content/drive/My Drive/Kaggle/Datasets/glove.840B.300d.pkl', allow_pickle=True)
fasttext_embeddings = np.load('/content/drive/My Drive/Kaggle/Datasets//crawl-300d-2M.pkl', allow_pickle=True)
```

```
CPU times: user 14.3 s, sys: 6.25 s, total: 20.6 s
Wall time: 1min 57s
```

```
def build_vocab(X):
    tweets = X.apply(lambda s: s.split()).values
    vocab = {}
    for tweet in tweets:
        for word in tweet:
            try:
                vocab[word] += 1
            except KeyError:
                vocab[word] = 1
    return vocab
def check_embeddings_coverage(X, embeddings):
     vocab = build_vocab(X)
    covered = {}
    oov = {}
    n_covered = 0
    n_oov = 0

    for word in vocab:
        try:
            covered[word] = embeddings[word]
            n_covered += vocab[word]
```

```
        except:
            oov[word] = vocab[word]
            n_oov += vocab[word]

    vocab_coverage = len(covered) / len(vocab)
    text_coverage = (n_covered / (n_covered + n_oov))
    sorted_oov = sorted(oov.items(), key=operator.itemgetter(1))[::-1]
    return sorted_oov, vocab_coverage, text_coverage

train_glove_oov, train_glove_vocab_coverage, train_glove_text_coverage = check_embeddings_coverage
(train_df['text'], glove_embeddings)
test_glove_oov, test_glove_vocab_coverage, test_glove_text_coverage = check_embeddings_coverage(te
st_df['text'], glove_embeddings)
print('GloVe Embeddings cover {:.2%} of vocabulary and {:.2%} of text in Training Set'.format(trai
n_glove_vocab_coverage, train_glove_text_coverage))
print('GloVe Embeddings cover {:.2%} of vocabulary and {:.2%} of text in Test Set'.format(test_glo
ve_vocab_coverage, test_glove_text_coverage))

train_fasttext_oov, train_fasttext_vocab_coverage, train_fasttext_text_coverage = check_embeddings
_coverage(train_df['text'], fasttext_embeddings)
test_fasttext_oov, test_fasttext_vocab_coverage, test_fasttext_text_coverage = check_embeddings_co
verage(test_df['text'], fasttext_embeddings)
print('FastText Embeddings cover {:.2%} of vocabulary and {:.2%} of text in Training Set'.format(t
rain_fasttext_vocab_coverage, train_fasttext_text_coverage))
print('FastText Embeddings cover {:.2%} of vocabulary and {:.2%} of text in Test Set'.format(test_
fasttext_vocab_coverage, test_fasttext_text_coverage))
```

```
GloVe Embeddings cover 52.06% of vocabulary and 82.68% of text in Training Set
GloVe Embeddings cover 57.21% of vocabulary and 81.85% of text in Test Set
FastText Embeddings cover 51.52% of vocabulary and 81.84% of text in Training
Set
FastText Embeddings cover 56.55% of vocabulary and 81.12% of text in Test Set
```

Words in the intersection of vocab and embeddings are stored in covered along with their counts. Words in vocab that don't exist in embeddings are stored in oov along with their counts. n_covered and n_oov are total number of counts and they are used for calculating coverage percentages. [6]

Both GloVe and FastText embeddings have more than 50% vocabulary and 80% text coverage without cleaning. GloVe and FastText coverage are very close but GloVe has slightly higher coverage. [6]

3.1.2 Cleaning

Tweets require lots of cleaning but it is inefficient to clean every single tweet because that would consume too much time. A general approach must be implemented for cleaning. [6]

- The most common type of words that require cleaning in oov have punctuations at the start or end. Those words don't have embeddings because of the trailing punctuations. Punctuations #, @, !, ?, +, &, -, $, =, <, >, |, {, }, ^, ', (,),[,], *, %, ..., ', ., :, ; are separated from words
- Special characters that are attached to words are removed completely
- Contractions are expanded

- Urls are removed
- Character entity references are replaced with their actual symbols
- Typos and slang are corrected, and informal abbreviations are written in their long forms
- Some words are replaced with their acronyms and some words are grouped into one
- Finally, hashtags and usernames contain lots of information about the context but they are written without spaces in between words so they don't have embeddings. Informational usernames and hashtags should be expanded but there are too many of them. I expanded as many as I could, but it takes too much time to run clean function after adding those replace calls.

Recalculate the embeddings coverage: -

```
GloVe Embeddings cover 82.89% of vocabulary and 97.14% of text in Training Set
GloVe Embeddings cover 88.09% of vocabulary and 97.32% of text in Test Set
FastText Embeddings cover 82.88% of vocabulary and 97.12% of text in Training Set
FastText Embeddings cover 87.80% of vocabulary and 97.25% of text in Test Set
```

3.1.3 Ambiguous Samples

As demonstrated earlier, it is observed that there are **18** unique tweets in training set which are classified differently in their duplicates. We have to manually reclassify them and rectify the error which leads to ambiguity.

The following table shows the new classes in which the tweets have been manually classified: -

text	New label
like for the music video I want some real action shit like burning buildings and police chases not some weak ben winston shit	0
Hellfire! We don‰Ûªt even want to think about it or mention it so let‰Ûªs not do anything that leads to it #islam!	0
The Prophet (peace be upon him) said 'Save yourself from Hellfire even if it is by giving half a date in charity.'	0
In #islam saving a person is equal in reward to saving all humans! Islam is the opposite of terrorism!	0
To fight bioterrorism sir.	0
Who is bringing the tornadoes and floods. Who is bringing the climate change. God is after America He is plaguing her\n \n#FARRAKHAN #QUOTE	1
#foodscare #offers2go #NestleIndia slips into loss after #Magginoodle #ban unsafe and hazardous for #humanconsumption	0
#Allah describes piling up #wealth thinking it would last #forever as the description of the people of #Hellfire in Surah Humaza. #Reflect	0
He came to a land which was engulfed in tribal war and turned it into a land of peace i.e. Madinah. #ProphetMuhammad #islam	0
RT NotExplained: The only known image of infamous hijacker D.B. Cooper. http://t.co/JlzK2HdeTG	1
Hellfire is surrounded by desires so be careful and don‰Ûªt let your desires control you! #Afterlife	0
CLEARED:incident with injury:I-495 inner loop Exit 31 - MD 97/Georgia Ave Silver Spring	1
Mmmmmm I'm burning.... I'm burning buildings I'm building.... Ooooooohhhh oooh ooh...	0
wowo--=== 12000 Nigerian refugees repatriated from Cameroon	0
.POTUS #StrategicPatience is a strategy for #Genocide; refugees; IDP Internally displaced people; horror; etc. https://t.co/rqWuoy1fm4	1
Caution: breathing may be hazardous to your health.	1
I Pledge Allegiance To The P.O.P.E. And The Burning Buildings of Epic City. ??????	0
that horrible sinking feeling when you‰Ûªve been at home on your phone for a while and you realise its been on 3G this whole time	0

3.2 Implementation

BERT Model Approach

3.2.1 Metrics

Keras has accuracy in its metrics module, but doesn't have rest of the metrics stated above. Another crucial point is **Precision**, **Recall** and **F1-Score** are global metrics so they should be calculated on whole training or validation set. Computing them on every batch would be both misleading and ineffective in terms of execution time. ClassificationReport which is similar to sklearn.metrics.classification_report, computes those metrics after every epoch for the given training and validation set. [6]

3.2.2 BERT LAYER

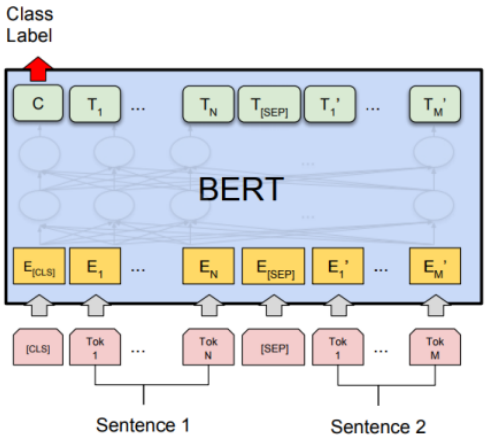
This model uses the implementation of BERT from the TensorFlow Models repository on GitHub at tensorflow/models/official/nlp/bert. It uses L=12 hidden layers (Transformer blocks), a hidden size of H=768, and A=12 attention heads.

This model has been pre-trained for English on the Wikipedia and Books Corpus. Inputs have been "uncased", meaning that the text has been lower-cased before tokenization into word pieces, and any accent markers have been stripped. We download the model for our use. [6]

```
bert_layer = hub.KerasLayer('https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/1', trainable=True)
```

3.2.3 ARCHITECTURE

BERT is a multi-layer bidirectional Transformer encoder. A word starts with its embedding representation from the embedding layer. Every layer does some multi-headed attention computation on the word representation of the previous layer to create a new intermediate representation. [4]



We write a wrapper class and include the necessary elements for the BERT model. It incorporates the cross-validation and metrics. For each fold of the cross validation, text is encoded and tokenized and fed to the BERT layer. Model is built and fit on the fold. Metrics are calculated and stored. [4]

The tokenization of input text is performed with the FullTokenizer class from *tensorflow/models/official/nlp/bert/tokenization.py.* [6]

max_seq_length parameter can be used for tuning the sequence length of text. Parameters such as lr, epochs and batch_size can be used for controlling the learning process. There are no dense or

pooling layers added after last layer of BERT. SGD is used as optimizer since others have hard time while converging. [6]

plot_learning_curve plots Accuracy, Precision, Recall and F1 Score (for validation set) stored after every epoch alongside with training/validation loss curve. This helps to see which metric fluctuates most while training. [6]

3.2.4 TRAINING EVALUATION AND PREDICTION

The operation logs for each fold for validation: -

```
Fold 0

Epoch 1/10
119/119 [==============================] - ETA: 0s - loss: 0.6401 - accuracy: 0.6469
Epoch: 1 - Training Precision: 0.732083 - Training Recall: 0.691128 - Training F1:
0.692759
Epoch: 1 - Validation Precision: 0.712058 - Validation Recall: 0.674732 - Validation
F1: 0.674957
119/119 [==============================] - 341s 3s/step - loss: 0.6401 - accuracy: 0.6469
- val_loss: 0.5969 - val_accuracy: 0.7024
Epoch 2/10
119/119 [==============================] - ETA: 0s - loss: 0.5402 - accuracy: 0.7470
Epoch: 2 - Training Precision: 0.772918 - Training Recall: 0.7512 - Training F1: 0.755852
Epoch: 2 - Validation Precision: 0.751364 - Validation Recall: 0.728843 - Validation
F1: 0.732768
119/119 [==============================] - 339s 3s/step - loss: 0.5402 - accuracy: 0.7470
- val_loss: 0.5299 - val_accuracy: 0.7476
Epoch 3/10
119/119 [==============================] - ETA: 0s - loss: 0.4784 - accuracy: 0.7782
Epoch: 3 - Training Precision: 0.795929 - Training Recall: 0.779324 - Training F1:
0.783888
Epoch: 3 - Validation Precision: 0.770813 - Validation Recall: 0.754026 - Validation
F1: 0.758105
119/119 [==============================] - 339s 3s/step - loss: 0.4784 - accuracy: 0.7782
- val_loss: 0.4919 - val_accuracy: 0.7691
Epoch 4/10
119/119 [==============================] - ETA: 0s - loss: 0.4401 - accuracy: 0.7993
Epoch: 4 - Training Precision: 0.817316 - Training Recall: 0.799417 - Training F1:
0.804487
Epoch: 4 - Validation Precision: 0.787586 - Validation Recall: 0.768965 - Validation
F1: 0.77355
119/119 [==============================] - 339s 3s/step - loss: 0.4401 - accuracy: 0.7993
- val_loss: 0.4691 - val_accuracy: 0.7841
Epoch 5/10
119/119 [==============================] - ETA: 0s - loss: 0.4145 - accuracy: 0.8179
Epoch: 5 - Training Precision: 0.824688 - Training Recall: 0.82056 - Training F1:
0.822308
Epoch: 5 - Validation Precision: 0.791523 - Validation Recall: 0.787842 - Validation
F1: 0.789373
119/119 [==============================] - 339s 3s/step - loss: 0.4145 - accuracy: 0.8179
- val_loss: 0.4577 - val_accuracy: 0.7949
Epoch 6/10
119/119 [==============================] - ETA: 0s - loss: 0.3973 - accuracy: 0.8271
Epoch: 6 - Training Precision: 0.838416 - Training Recall: 0.828171 - Training F1:
0.831839
Epoch: 6 - Validation Precision: 0.803765 - Validation Recall: 0.794752 - Validation
F1: 0.797873
119/119 [==============================] - 339s 3s/step - loss: 0.3973 - accuracy: 0.8271
- val_loss: 0.4458 - val_accuracy: 0.8046
Epoch 7/10
119/119 [==============================] - ETA: 0s - loss: 0.3822 - accuracy: 0.8376
Epoch: 7 - Training Precision: 0.852943 - Training Recall: 0.836871 - Training F1:
0.842015
Epoch: 7 - Validation Precision: 0.810993 - Validation Recall: 0.796068 - Validation
F1: 0.800475
119/119 [==============================] - 339s 3s/step - loss: 0.3822 - accuracy: 0.8376
- val_loss: 0.4395 - val_accuracy: 0.8085
Epoch 8/10
```

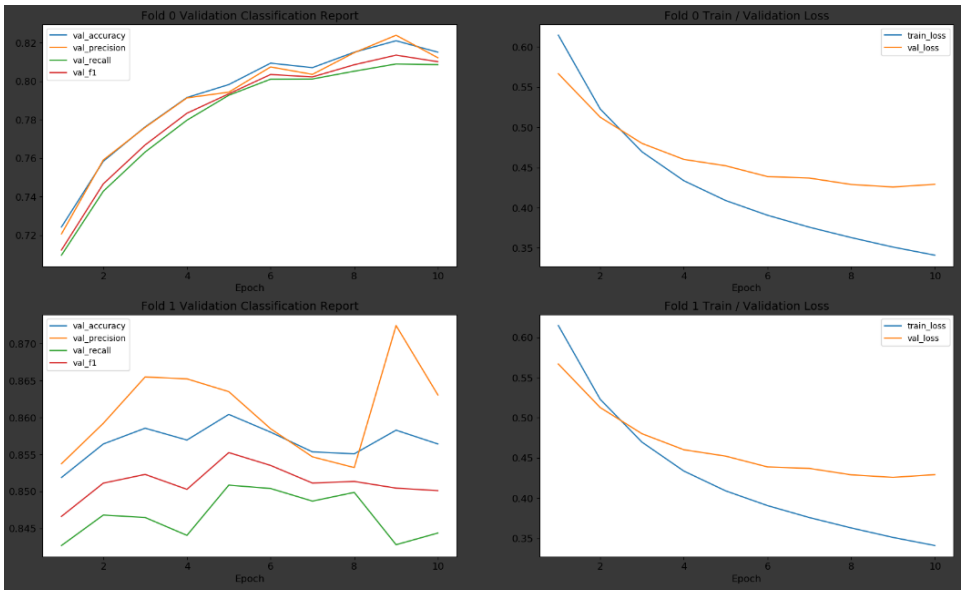
```
119/119 [==============================] - ETA: 0s - loss: 0.3689 - accuracy: 0.8463
Epoch: 8 - Training Precision: 0.858285 - Training Recall: 0.84804 - Training F1:
0.851806
Epoch: 8 - Validation Precision: 0.810306 - Validation Recall: 0.800727 - Validation
F1: 0.804024
119/119 [==============================] - 339s 3s/step - loss: 0.3689 - accuracy: 0.8463
- val_loss: 0.4345 - val_accuracy: 0.8106
Epoch 9/10
119/119 [==============================] - ETA: 0s - loss: 0.3578 - accuracy: 0.8497
Epoch: 9 - Training Precision: 0.861983 - Training Recall: 0.853966 - Training F1:
0.857093
Epoch: 9 - Validation Precision: 0.814235 - Validation Recall: 0.806085 - Validation
F1: 0.809039
119/119 [==============================] - 339s 3s/step - loss: 0.3578 - accuracy: 0.8497
- val_loss: 0.4322 - val_accuracy: 0.8151
Epoch 10/10
119/119 [==============================] - ETA: 0s - loss: 0.3471 - accuracy: 0.8584
Epoch: 10 - Training Precision: 0.868335 - Training Recall: 0.859095 - Training F1:
0.862615
Epoch: 10 - Validation Precision: 0.815957 - Validation Recall: 0.807617 - Validation
F1: 0.81063
119/119 [==============================] - 339s 3s/step - loss: 0.3471 - accuracy: 0.8584
- val_loss: 0.4309 - val_accuracy: 0.8167

Fold 1

Epoch 1/10
119/119 [==============================] - ETA: 0s - loss: 0.5639 - accuracy: 0.7050
Epoch: 1 - Training Precision: 0.80567 - Training Recall: 0.797812 - Training F1:
0.800649
Epoch: 1 - Validation Precision: 0.83151 - Validation Recall: 0.824751 - Validation F1:
0.827395
119/119 [==============================] - 341s 3s/step - loss: 0.5639 - accuracy: 0.7050
- val_loss: 0.4155 - val_accuracy: 0.8326
Epoch 2/10
119/119 [==============================] - ETA: 0s - loss: 0.4326 - accuracy: 0.8098
Epoch: 2 - Training Precision: 0.833036 - Training Recall: 0.812087 - Training F1:
0.817758
Epoch: 2 - Validation Precision: 0.851531 - Validation Recall: 0.832014 - Validation
F1: 0.837816
119/119 [==============================] - 338s 3s/step - loss: 0.4326 - accuracy: 0.8098
- val_loss: 0.3859 - val_accuracy: 0.8450
Epoch 3/10
119/119 [==============================] - ETA: 0s - loss: 0.4071 - accuracy: 0.8266
Epoch: 3 - Training Precision: 0.836267 - Training Recall: 0.823024 - Training F1:
0.827344
Epoch: 3 - Validation Precision: 0.855804 - Validation Recall: 0.841333 - Validation
F1: 0.846149
119/119 [==============================] - 338s 3s/step - loss: 0.4071 - accuracy: 0.8266
- val_loss: 0.3744 - val_accuracy: 0.8521
Epoch 4/10
119/119 [==============================] - ETA: 0s - loss: 0.3914 - accuracy: 0.8340
Epoch: 4 - Training Precision: 0.853595 - Training Recall: 0.824967 - Training F1:
0.832027
Epoch: 4 - Validation Precision: 0.858838 - Validation Recall: 0.83004 - Validation F1:
0.837378
119/119 [==============================] - 338s 3s/step - loss: 0.3914 - accuracy: 0.8340
- val_loss: 0.3768 - val_accuracy: 0.8460
Epoch 5/10
119/119 [==============================] - ETA: 0s - loss: 0.3765 - accuracy: 0.8432
Epoch: 5 - Training Precision: 0.844461 - Training Recall: 0.840768 - Training F1:
0.842368
Epoch: 5 - Validation Precision: 0.846504 - Validation Recall: 0.841336 - Validation
F1: 0.843488
119/119 [==============================] - 338s 3s/step - loss: 0.3765 - accuracy: 0.8432
- val_loss: 0.3718 - val_accuracy: 0.8479
Epoch 6/10
119/119 [==============================] - ETA: 0s - loss: 0.3642 - accuracy: 0.8500
Epoch: 6 - Training Precision: 0.860787 - Training Recall: 0.844857 - Training F1:
0.849969
Epoch: 6 - Validation Precision: 0.858384 - Validation Recall: 0.841469 - Validation
F1: 0.846842
119/119 [==============================] - 338s 3s/step - loss: 0.3642 - accuracy: 0.8500
- val_loss: 0.3649 - val_accuracy: 0.8531
Epoch 7/10
119/119 [==============================] - ETA: 0s - loss: 0.3521 - accuracy: 0.8519
```

```
Epoch: 7 - Training Precision: 0.861023 - Training Recall: 0.85166 - Training F1:
0.855151
Epoch: 7 - Validation Precision: 0.853101 - Validation Recall: 0.841895 - Validation
F1: 0.845894
119/119 [==============================] - 338s 3s/step - loss: 0.3521 - accuracy: 0.8519
- val_loss: 0.3636 - val_accuracy: 0.8513
Epoch 8/10
119/119 [==============================] - ETA: 0s - loss: 0.3422 - accuracy: 0.8597
Epoch: 8 - Training Precision: 0.864705 - Training Recall: 0.854572 - Training F1:
0.858296
Epoch: 8 - Validation Precision: 0.853831 - Validation Recall: 0.841812 - Validation
F1: 0.846025
119/119 [==============================] - 338s 3s/step - loss: 0.3422 - accuracy: 0.8597
- val_loss: 0.3635 - val_accuracy: 0.8516
Epoch 9/10
119/119 [==============================] - ETA: 0s - loss: 0.3334 - accuracy: 0.8595
Epoch: 9 - Training Precision: 0.871817 - Training Recall: 0.859708 - Training F1:
0.863999
Epoch: 9 - Validation Precision: 0.854569 - Validation Recall: 0.840801 - Validation
F1: 0.845445
119/119 [==============================] - 338s 3s/step - loss: 0.3334 - accuracy: 0.8595
- val_loss: 0.3628 - val_accuracy: 0.8513
Epoch 10/10
119/119 [==============================] - ETA: 0s - loss: 0.3221 - accuracy: 0.8673
Epoch: 10 - Training Precision: 0.874052 - Training Recall: 0.866667 - Training F1:
0.869601
Epoch: 10 - Validation Precision: 0.849444 - Validation Recall: 0.840998 - Validation
F1: 0.844213
119/119 [==============================] - 338s 3s/step - loss: 0.3221 - accuracy: 0.8673
- val_loss: 0.3658 - val_accuracy: 0.8492
```

Learning Curve



After this we use the trained model to predict the classes for the test data set.

TFIDF Approach

3.2.5 TFIDF Vectorization

We create the TF-IDF vector using the following code:-

```
tfidf_vec = TfidfVectorizer(stop_words='english', ngram_range=(1,3))
tfidf_vec.fit_transform(train_df['text_cleaned'].values.tolist() + te
st_df['text_cleaned'].values.tolist())
train_tfidf = tfidf_vec.transform(train_df['text_cleaned'].values.tol
ist())
test_tfidf = tfidf_vec.transform(test_df['text_cleaned'].values.tolis
t())
```

3.2.6 Building Models using default parameters

We use classifiers with their default parameters and observe which classifiers work well with the data present. It helps us to prioritize the models to be chosen for tuning.

The results of the evaluation are as follows: -

```
Evaluation started for  LogisticRegression
Average accuracy of LogisticRegression is 74.85%
```

```
Evaluation started for  KNN
Average accuracy of KNN is 75.71%
```

```
Evaluation started for  Kernel SVM
Average accuracy of Kernel SVM is 73.08%
```

```
Evaluation started for  RandomForest
Average accuracy of RandomForest is 75.77%
```

```
Evaluation started for  XGBoost
Average accuracy of XGBoost is 70.78%
```

The classifier with the highest accuracy is Random Forest classifier. The sequence of classifier performance from best to worst is as follows: -

Classifier	**Accuracy**
Random Forest	75.77%
KNN	75.71%
Logistic Regression	74.85%
Kernel SVM	73.08%
Decision Tree	73.01%
XGBoost	70.78%

3.3 Refinement

Model Tuning

We perform hyperparameter tuning for each classifier one by one and check the accuracy rise and drop. For this specific purpose, we use Grid Search to get the optimal parameters i.e. the hyperparameters which give the best accuracy of the lot.

Results of Grid Search is as follows: -

Random Forest

```
Best Accuracy: 76.58%
Best Parameters: {'n_estimators': 150}
```

Logistic Regression

```
Best Accuracy: 79.67%
Best Parameters: {'C': 100, 'penalty': 'l1', 'solver': 'saga'}
```

KNN

```
Best Accuracy: 77.54%
Best Parameters: {'leaf_size': 30, 'metric': 'minkowski', 'n_neighbors': 43}
```

SVC

```
Best Accuracy: 79.59%
Best Parameters: {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}
```

XGBoost

```
Best Accuracy: 71.52%
Best parameters: {'subsample': 0.6, 'min_child_weight': 1, 'max_depth': 5,
'gamma': 1.5, 'colsample_bytree': 0.8}
```

4. Results

4.1 Model Evaluation and Validation

4.1.1 BERT Model Approach

As explained previously, the BERT Model was trained and we used it to predict finally on the data set.

The predictions for the test dataset were submitted to the Kaggle portal for evaluation.

The results are as follows: -

The accuracy achieved by the model was 82.041 %

4.1.2 TFIDF Approach

As explained previously, the TFIDF approach gave us a sparse matrix of words. We trained different classifiers on that matrix. We used it to predict finally on the test data set.

The predictions of the different classifiers for the test dataset were submitted to the Kaggle portal for evaluation.

The results are as follows: -

Classifier	**Accuracy**
Random Forest	77.51%
KNN	77.11%
Logistic Regression	79.93%
Kernel SVM	79.99%
XGBoost	73.95%

4.2 Justification

The performance of all the classifiers used to predict on top of the TF-IDF vector is more or less the same, with the Kernel SVM Classifier showing the best results with about 80% accuracy.
The major time-consuming part in this approach was the tuning of the hyperparameters of the individual models. Rest of the operations consumed negligible time.

The performance of the BERT model is superior with 82.041% of accuracy.
The time however taken to train the BERT model was higher. I was able to use the GPU provided by Google Colab, which accelerated the hardware, otherwise, it would have required even more time. It consists of a deep network therefore, it is understandable.

As a final verdict, a faster and agile model is always better but accuracy in this type of problem is also of high importance. Since, the problem is related to the disaster tweets, it is crucial and every single rise in accuracy counts. Furthermore, with an even larger dataset, in a real-world situation, the BERT model would just get better and better. So, I would go with the BERT model.

REFERENCES

1. Kaggle Competition: Real or Not? https://www.kaggle.com/c/nlp-getting-started/
2. Text Mining: The Beginner's Guide, https://monkeylearn.com/text-mining/
3. 'What are N-Grams?', by Kavita Ganesan, n.p. https://kavita-ganesan.com/what-are-n-grams/
4. 'BERT Explained – A list of Frequently Asked Questions', by Yashu Seth (2019, June 12) n.p. https://yashuseth.blog/2019/06/12/bert-explained-faqs-understand-bert-working/
5. 'TF-IDF/Term Frequency Technique: Easiest explanation for Text classification in NLP using Python (Chatbot training on words)' by Rohit Madan, (2019, May 31) https://medium.com/analytics-vidhya/tf-idf-term-frequency-technique-easiest-explanation-for-text-classification-in-nlp-with-code-8ca3912e58c3
6. 'Kaggle Kernel: LP with Disaster Tweets - EDA, Cleaning and BERT' by Gunes Evitan (July 2020) n.p. https://www.kaggle.com/gunesevitan/nlp-with-disaster-tweets-eda-cleaning-and-bert
7. 'Bert – Explained state of art language model for NLP', Medium by Rani Horev (2018, November 10) n.p. https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270
8. 'Understanding BERT' by Kirti Vashee (2020, June 26) n.p. https://blog.sdl.com/blog/Understanding-BERT.html
9. 'Demystifying BERT: A Comprehensive Guide to the Groundbreaking NLP Framework' by Mohd Sanad Zaki Rizvi (2019, September 25) n.p. https://www.analyticsvidhya.com/blog/2019/09/demystifying-bert-groundbreaking-nlp-framework/
10. Library description, TFIDF vectorizer: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
11. 'What Are Word Embeddings for Text?' by Jason Brownlee (2017, October 11) n.p. https://machinelearningmastery.com/what-are-word-embeddings/