Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Master's diploma thesis

in the field of study Computer Science
and specialisation Artificial Intelligence

Applying machine learning to disaster detection in Twitter tweets

Maryan Plakhtiy
student record book number 288058

thesis supervisor
dr hab. Marcin Paprzycki

WARSAW 2020

....	....
supervisor's signature	author's signature

Abstract

Applying machine learning to disaster detection in Twitter tweets

Nowadays, disaster detection, based on Twitter tweets, has become one of the challenging and demanded researches. For that purpose there was even a competition, created in the Kaggle repository. In this work we explore (and compare) various machine learning methods and techniques, applying them to the disaster tweets data set from the Kaggle competition. We select 10 most promising data preprocessing algorithms, on which our models and classifiers are tested. We find the most successful pairs of model/classifier and preprocessing algorithm for further use. Exploring an approach of combining predictions from different sources to produce a finer predictions, we train several simple neural network models, on data built from combinations of best pairs' predictions. We compare the achieved results to the results obtained when applying simple formulas, to the same combinations of results from the original classifiers. Finally, we locate tweets, which were not recognized by any model or classifier and remove them from the training data set, analysing how this action influences the performance of models and classifiers.
Keywords: Disaster detection, NLP, Twitter

Streszczenie

Zastosowanie technik uczenia maszynowego do wykrywania katastrof na podstawie postów w serwisie Twitter

Wykrywanie katastrof na podstawie tweetów na Twitterze stało się obecnie jednym z trudnych i zapotrzebowanych badań. W tym celu powstał nawet konkurs, utworzony w repozytorium Kaggle. W tej pracy badamy (i porównujemy) różne metody uczenia maszynowego i techniki ulepszenia ich wyników przy użyciu zestawu danych tweetów katastroficznych z konkursu Kaggle. Wybieramy 10 najbardziej obiecujących algorytmów wstępnego przetwarzania danych, na których testowane są nasze modele i klasyfikatory. Znajdujemy najbardziej udane pary model / klasyfikator i algorytm wstępnego przetwarzania do dalszego wykorzystania. Badając podejście polegające na łączeniu prognoz z różnych źródeł w celu uzyskania dokładniejszych prognoz, trenujemy kilka prostych modeli sieci neuronowych na danych zbudowanych z kombinacji przewidywań najlepszych par. Uzyskane wyniki porównujemy z wynikami uzyskanymi przy zastosowaniu prostych formuł, do tych samych kombinacji wyników z oryginalnych klasyfikatorów. Na koniec lokalizujemy tweety, które nie zostały rozpoznane przez żaden model ani klasyfikator, i usuwamy je ze zbioru danych uczących, analizując, jak ta akcja wpływa na wydajność modeli i klasyfikatorów.

Słowa kluczowe: Wykrywanie katastrof, NLP, Twitter

Warsaw, ...

Declaration

I hereby declare that the thesis entitled „Applying machine learning to disaster detection in Twitter tweets”, submitted for the Master degree, supervised by dr hab. Marcin Paprzycki, is entirely my original work apart from the recognized reference.

....

Contents

1. Introduction

Nowadays, social media like Facebook, Twitter, Instagram, and others, are one of the most followed sources of news. Almost all news agencies, organizations, and others, post their information/news/advertisements there, as they are the most visited web pages, every day. It can be said that each of these social media outlets has its own purpose, or target audience that they are best suited for. For example, Facebook is known to be a political discussion arena, Instagram is commonly used by bloggers and businesses to advertise their products. Twitter, because of its short and informative messages, has become one of the fastest, and widely used, news/events spreader, and the most important source of communication in case of emergency. Especially nowadays, when almost every person has a smartphone, which lets them announce an emergency that they are observing in the real-time. Because of this, there are more and more news agencies, emergency organizations and others, who are interested in monitoring Twitter for emergency announcements. Successful monitoring would help to reply and act on emergencies faster and could save lives.

However, the most challenging question is, how to distinguish an actual emergency-announcing tweet, from the fake news, clickbaits, or non-related tweets. Most of these tweets can be easily identified by a human, but it is quite challenging for a machine. Event detection, on the basis of tweets' content, is a popular research area in data analytics. While the most popular among researchers seems to be the sentiment analysis, there are also other areas. For example, fake news detection, text classification, or disaster detection.

Disaster detection is one of the challenging and demanding areas, these days. For this purpose there was a competition created on the Kaggle [10] (popular online community of data scientists and machine learning practitioners), which offered 10000$ for the competition's winner, who would provide a solution that would be able to classify correctly the biggest amount of tweets. Unfortunately, during the work on this topic, the competition prize was canceled, due to the data leakage of correct predictions for the test data set.

This work is based on the data set taken from this competition, and its aim is to apply broadly understood machine learning tools and methods, to analyse content of tweets and to detect disasters. One of the directions that will also be considered, is combining results from multiple classifiers and models for training on neural network(s). It is expected that this approach will help to obtain better predictions than from separately used classifiers and models.

1.1. State-of-the-Art Analysis

In this section, the state-of-the-art machine learning architectures candidates, for the disaster detection are described. The main purpose of the section is to illustrate the most recent, modern and successful approaches that are worthy further examination.

In [29], which is one of the key related research papers, authors decided to leverage a Hybrid CNN-LSTM network architecture for their Fire Burst Detection task. Unfortunately, they do not provide achieved accuracy for this network, but they claim it was successful for their mission. This network was chosen based on the [1] paper. There, its authors, for the Fake News Identification on Twitter purpose, tested LSTM, LSTM-Dropout and Hybryd CNN-LSTM networks. They achieved 82.29% accuracy with the LSTM architecture, 73.78% with the LSTM-Dropout and 80.38% with Hybryd CNN-LSTM on the PHEME data set. The PHEME data set consisted of approximately 5,800 tweets involving five rumor stories. Kaggle data set used in our work is bigger, and it was expected that this can improve these models accuracy. These models are promising to our research, as the data set consist of different kind of disasters and these models work without the context. However, our worry was also that larger dataset may involve more "noise" (tweets about 'anything") and thus achieving high accuracy may be complex.

In [2], authors had a quite similar aim. They analysed damage caused by the disaster (how many people were injured, homes destroyed, etc). They have created a Tweedr (Twitter for Disaster Response) application, which consists of three main parts: classification, clustering and extraction. Similarly to our goal, they had a task to identify disaster tweets first. For this purpose they used a small subset of 1049 of all tweets, where 793 were labeled as positive examples. They used popular classifiers, where Logistic Regression was proven to be the most effective, with the 88% accuracy for the disaster detection. Unfortunately, its hard to say if this method would prove the same success in our work, because data set on which these classifiers were tested was really small and had more positive examples. Again, large data set may have its advantages and disadvantages, depending on the intrinsic properties of the data itself.

In [14], one of the resourceful papers on techniques and applications for Data Analysis on Twitter, authors prepared a table with the effective techniques for each of Data Analysis areas. They also pointed out to the fact that Logistic Regression is best suited for the disaster management applications. This paper supported leveraging old and well-known classifiers, thus some of them will be considered for our purpose.

In [30], authors proposed Hierarchical Attention Networks (HAN) for Document Classification, which outperformed other baseline models and linear methods. They compared several popular methods, testing them on 6 different classification tasks, and stated that HAN architecture won on each of them. In this paper, authors worked with whole documents, whereas in the case of Twitter, a text entry is limited to max 140 chars (and some entries do not use this maximum size, but are shorter). Hopefully, this network will show similar performance in our case. Another paper in favor of usage of the attention mechanism is [31], where authors solve hyperpartisan news detection problem. Hyperpartisan news are such news, which are highly polarized, extremely biased for or against a given party. Authors compared different networks models on a big data set (1 million articles), and claimed that the Bidirectioanal LSTM model with self attention (Bi-LSTM-ATTN) had the best performance. Similarly to [30], their worked with the big text entries, but they reduced them to 392 tokens, during the preprocessing phase, removing stopwords and selecting only 40000 of most popular tokens vocabulary. With these approach and using the Bi-LSTM-ATTN, they were able to obtain 93,68% accuracy for their hyperpartisan news detection problem, while other models obtained around 89,77% - 91,74%. Mentioned model, and preprocessing approach, are promising for our mission. However, removing stopwords, in case of tweets in preprocessing phase, might drop valuable information taking into account that tweets are limited to 140 characters. This model will be tested additionally if HAN [30] model will prove successful enough as they both are based on the attention approach.

One of the cutting-edge approaches is using BERT (Bidirectional Encoder Representations from Transformers [3], [7]), which is a new method of pre-training language representations, which obtains state-of-the-art results on a wide array of Natural Language Processing (NLP) tasks. The author, in [15], used BERT for the tweets classification problem in the field of disaster management. In the work, the BERT-default and its modifications were compared to the baseline model. These models were tested on the combination of CrisisLex and some CrisisNLP data sets. The resulted data set consisted of 75800 labeled tweets, and had such labels: affected individuals, infrastructure and utilities damage, caution and advice, donation and volunteering, sympathy and emotional support, other useful information, not related or not informative, etc. All BERT models outperformed the baseline one, and the BERT-default had the highest recall score among others. This approach was proven to be very effective in the NLP problems. Thus it will be tested in our work.

Last, but not least, inspiring papers are [6] and [28] where authors combined information from multiple search engines in order to produce a better result query. In these papers they looked into three approaches, based on: Game Theory, Auction and Consensus methods. As a result, each of these methods proved to have own advantages and features that can be utilized. The idea of results combination looks very promising to our work as we will have multiple models and classifiers (sources) tested. Thus this approach can be applied for our set of tested models. It is obvious that different models and classifiers may produce unique results, or in other words – unique probabilities of a tweet being a disaster tweet or not. It is expected that combination of such predicted probabilities can be trained on a neural network and its score may outperform the score of single models.

In all of the above researches it is barely touched the data preprocessing phase. In this work, besides testing known models' architectures and approaches, each model will be tested on several preprocessing algorithms. It is hoped to find the best and universal preprocessing algorithm, which can be used for all of the models.

To sum up, in this work it was decided to test six classifiers and eleven neural network models. The six classifiers are: Logistic Regression, Ridge, SVC, SGD, Decision Tree and Random Forest. The eleven neural networks models are: LSTM, Bi-LSTM, LSTM-Dropout, CNN-LSTM, Fast Text, RNN, RCNN, CNN, GRU, HAN and BERT. The classifier will be tested using Count [24] and Tfidf [27] vectorizers. Ten neural networks models (excluding BERT) will be tested using Keras Embeddings [11] and Global Vectors for Word Representation (GloVe [19]) Embeddings. Each of these models will be tested together with several selected preprocessing algorithms. Afterwards, combining results from multiple classifiers and models for training on neural network will be tried to obtain better predictions.

1.2. Data Set and its Analysis

Let us now describe in some detail the Kaggle data set that was used in our work.

1.2.1. Data Set

Data set is taken from the Kaggle competition: *"Real or Not? NLP with Disaster Tweets. Predict which Tweets are about real disasters and which ones are not"* [10]. The complete data set contains 10,876 tweets that were hand classified into two categories. The first category is the *True* tweets that notify about the real disaster, and the second category of the *False* tweets. This data set was (within Kaggle) divided into two files: *train.csv* and *test.csv.* Additionally, for those who would like to participate in the competition, Kaggle provides a *sample_submission.csv* file, which is to be used when submitting the obtained results.

The first file – *train.csv* – contains data, on which one is expected to train models, classifiers, etc. This csv file contains 7613 rows, and 5 columns: *id*, *keyword*, *location*, *text*, *target*. First 10 rows from the *train.csv* file are presented in the table 1.1.

Table 1.1: First 10 rows from train.csv file.

id	keyword	location	text	target
1	NaN	NaN	Our Deeds are the Reason of this #earthquake M...	1
4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
5	NaN	NaN	All residents asked to 'shelter in place' are ...	1
6	NaN	NaN	13,000 people receive #wildfires evacuation or...	1
7	NaN	NaN	Just got sent this photo from Ruby #Alaska as ...	1
8	NaN	NaN	#RockyFire Update => California Hwy. 20 closed...	1
10	NaN	NaN	#flood #disaster Heavy rain causes flash flood...	1
13	NaN	NaN	I'm on top of the hill and I can see a fire in...	1
14	NaN	NaN	There's an emergency evacuation happening now ...	1
15	NaN	NaN	I'm afraid that the tornado is coming to our a...	1

Individual columns from the table 1.1 have the following meaning:

- *id* column does not contain any relevant information so it will not be used in any way,
- *keyword* column contains keywords from tweets or *NaN* if tweet does not have a keyword word,
- *location* column contains location from tweets or *NaN* if tweet does not have a location provided,
- *text* column contains tweets texts,
- *target* column contains correct predictions for tweets. *1* means that tweet is recognized as one which notifies about the disaster, *0* – that tweet does not announce an emergency.

The second file *test.csv* contains data, which should be used to test the trained model. Based on tweets predictions, obtained on the basis of applying models to this file, the score will be calculated, and placed in the Kaggle Leaderboard [9]. This file contains 3263 rows and 4 columns: *id*, *keyword*, *location* and *text*. First 10 rows from *test.csv* file are presented in the table 1.2. Obviously, this file does not contain the *target* column.

Table 1.2: First 10 rows from test.csv file.

id	keyword	location	text
0	NaN	NaN	Just happened a terrible car crash
2	NaN	NaN	Heard about #earthquake is different cities, s...
3	NaN	NaN	there is a forest fire at spot pond, geese are...
9	NaN	NaN	Apocalypse lighting. #Spokane #wildfires
11	NaN	NaN	Typhoon Soudelor kills 28 in China and Taiwan
12	NaN	NaN	We're shaking...It's an earthquake
21	NaN	NaN	They'd probably still show more life than Arse...
22	NaN	NaN	Hey! How are you?
27	NaN	NaN	What a nice hat?
29	NaN	NaN	Fuck off!

The third file *submisson.csv* contains example how the submission file to Kaggle should look like. First 5 rows from this file are presented in the table 1.3.

Table 1.3: First 5 rows from submisson.csv file.

id	target
0	0
2	0
3	0
9	0
11	0

Following the link to the Kaggle Leaderboard [9], it is easy to notice that a lot of competitors in this competition have a perfect score. This is because of the data leakage for this competition [1]. In this context, it has been established that competitors who have a perfect score used the leaked test.csv file (with correct predictions) in their training process. There is also a valid claim that some participants have simply submitted the correct predictions to Kaggle, without creating and training a model. Moreover, it is **not possible** to establish, which results that are not 100% accurate have been actually obtained by training models. This is because it is easy

[1]Data leak means that correct predictions for tweets from test file can be found in the Internet

to envision that some participants decided to “adapt” their results so that the will be close to perfect, but not “stupidly perfect”.

Unfortunately, this data leakage had place after the work on this competition was started. Before it was noticed by the Kaggle there was a prize of 10000$ for the competition’s winner with the best result. Hence, Kaggle had to cancel this competition as it was impossible to choose the winner if everybody could used the leaked data to increase their performance.

Overall, analyzing the leaderboard table it can be assumed that the first valid result might be 0.86607 submitted by the "right_right_team" team. However, for the reasons specified above, this claim is a “reasonable speculation”. In the table 1.4 there are top 10 results from the Kaggle leaderboard which are not grater than 0.98. It is very likely that the actual best result is among these.

Table 1.4: First 10 competitors results

TeamId	TeamName	SubmissionDate	Score
5048476	right_right_team	2020-06-21	0.86607
5048476	right_right_team	2020-06-16	0.85534
4921973	LucasLau	2020-05-20	0.85473
5163136	Mitten	2020-07-03	0.85136
4923119	Carmen@Sandiego	2020-05-21	0.84983
4921973	LucasLau	2020-05-20	0.84799
4906383	Wei Shih Tu	2020-05-21	0.84799
5153352	Naman Sood	2020-06-29	0.84768
4787020	Tran Duc Manh	2020-07-08	0.84737
5048476	right_right_team	2020-06-16	0.84707

In this work, it was decided to create *test_with_target.csv* file, which is the same as *test.csv* file, but it has *target* column with correct predictions from leaked data. This file was created in order to simplify calculation of the Kaggle Leaderboard score. Without this file it would be needed to submit a submission file to Kaggle every time when there is a need to find an actual score of the model. In other words, this file is a purely technical way of being able to compare the result with these reported in the Leaderboard of the Kaggle competition. The first 10 rows from this file presented in the table 1.5.

Table 1.5: First 10 rows from test_with_target.csv file.

id	keyword	location	text	target
0	NaN	NaN	Just happened a terrible car crash	1
1	NaN	NaN	Heard about #earthquake is different cities, s...	1
2	NaN	NaN	there is a forest fire at spot pond, geese are...	1
3	NaN	NaN	Apocalypse lighting. #Spokane #wildfires	1
4	NaN	NaN	Typhoon Soudelor kills 28 in China and Taiwan	1
5	NaN	NaN	We're shaking...It's an earthquake	1
6	NaN	NaN	They'd probably still show more life than Arse...	0
7	NaN	NaN	Hey! How are you?	0
8	NaN	NaN	What a nice hat?	0
9	NaN	NaN	Fuck off!	0

To sum up, unless otherwise stated (see, section 3.10), models and classifiers will be trained and validated using data only from the *train.csv* file. While the corresponding Kaggle official score will be calculated using data from the *test_with_target.csv* file.

In this work, models and classifiers will be evaluated using standard 10-fold cross-validation approach. This means that data will be divided into ten equal parts, where nine will be used for training and one – for validation on the each fold. As a result each model and classifier will be the trained on the *Train Data Set* (90%) and validated on the *Validation Data Set* (10%) from *train.csv* file on each fold. Additionally, Kaggle score will be calculated with the *Test Data Set* from the *test_with_target.csv* file on the each fold.

1.2.2. Data Set Analysis

The Python program, inspired by [4], was developed to analyze data we are dealing with. Let's consider the *Train Data Set* from the *train.csv* file, and the *Test Data Set* from the *test.csv* file. All of the data features (class distribution, characters distribution, etc) have very similar trends, in the Train and the Test data sets. This proves that both of these data sets are taken from the same sample.

In the figures 1.1 and 1.2 there are class distributions for the *Train Data Set* and the *Test Data Set* respectively. It is easily seen that both figures have similar trends. In both data sets there are more non disaster tweets then disaster tweets.

Figure 1.1: Class Distribution Chart for the Train Data Set

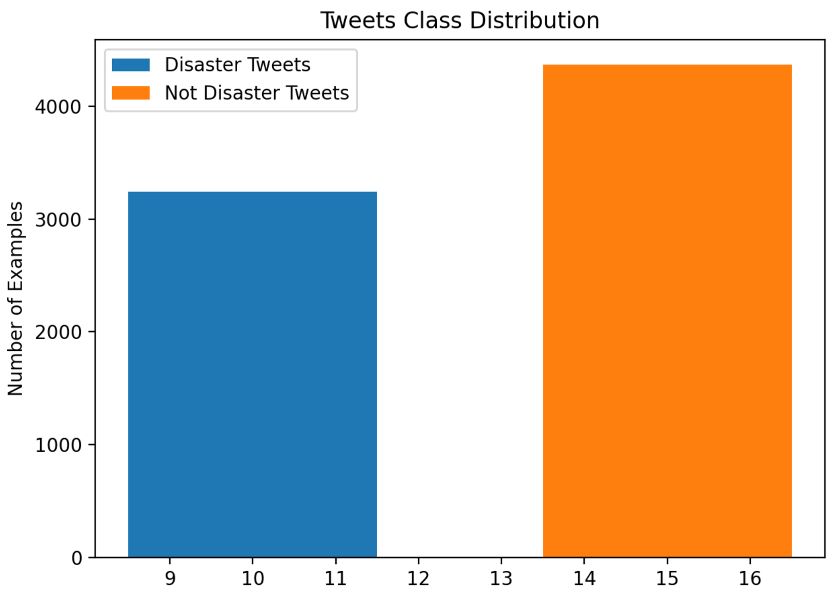
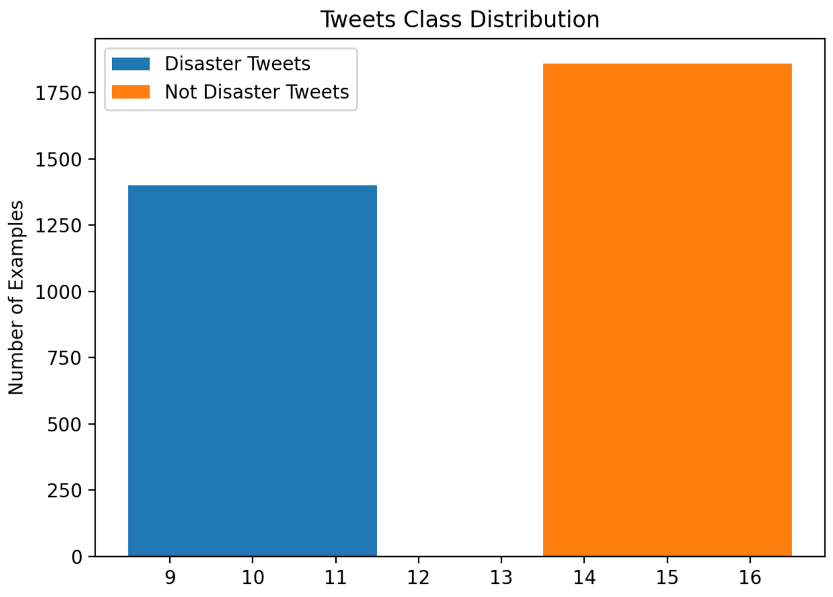


Figure 1.2: Class Distribution Chart for the Test Data Set



Each of data sets, from the csv files has a *keyword* column. In figures 1.3 and 1.4 shown is how many tweets of each class have a keyword data. It can be noticed that trends in these two figures match trends of class distribution figures (1.1, 1.2). It thus can be concluded that there is no visible correlation between tweet keyword data and its class. Nevertheless, there will be a possibility to include the presence of the keyword data in the training process.

Figure 1.3: Keyword Column Distribution Chart for the Train Data Set

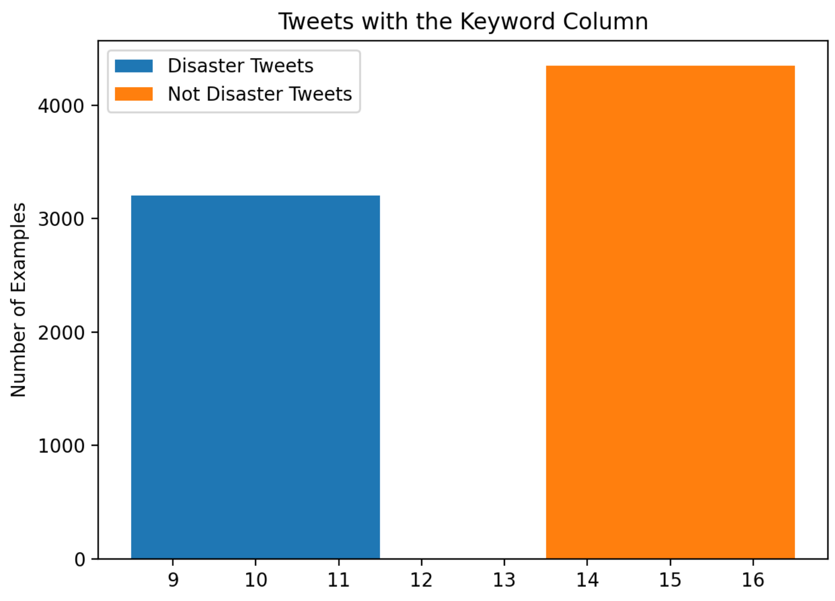
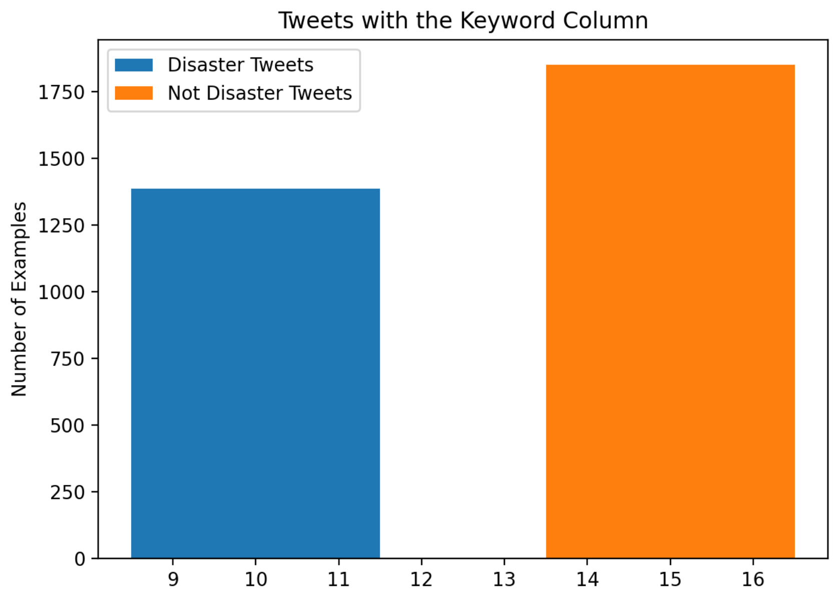


Figure 1.4: Keyword Column Distribution Chart for the Test Data Set



In the figures 1.5 and 1.6 it is visible that the considerable number of tweets do not have the location data but still general trend is very similar to the class distribution figures (1.1, 1.2). The same as for the location figures, we can conclude that there is no visible correlation between tweet location data and its class. But the training will be conducted with and without this information to check this conclusion.

Figure 1.5: Location Column Distribution Chart for the Train Data Set

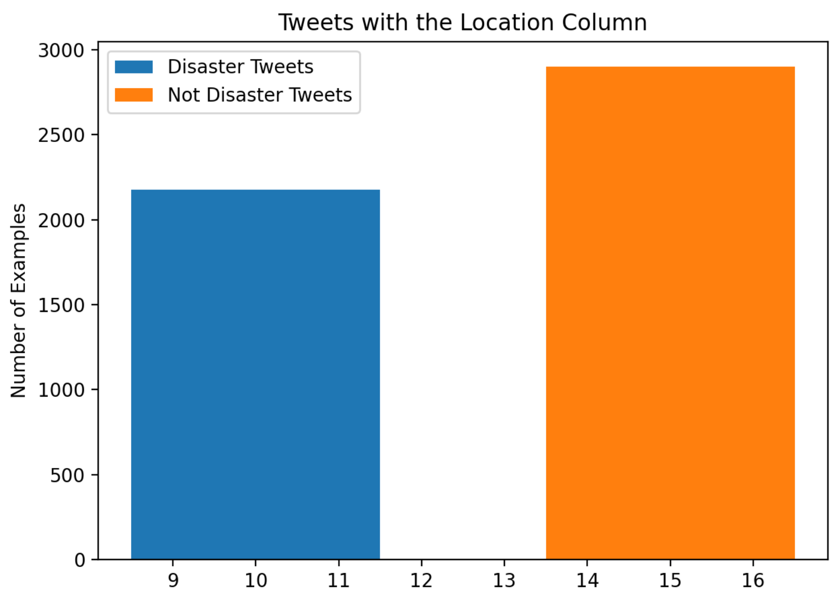
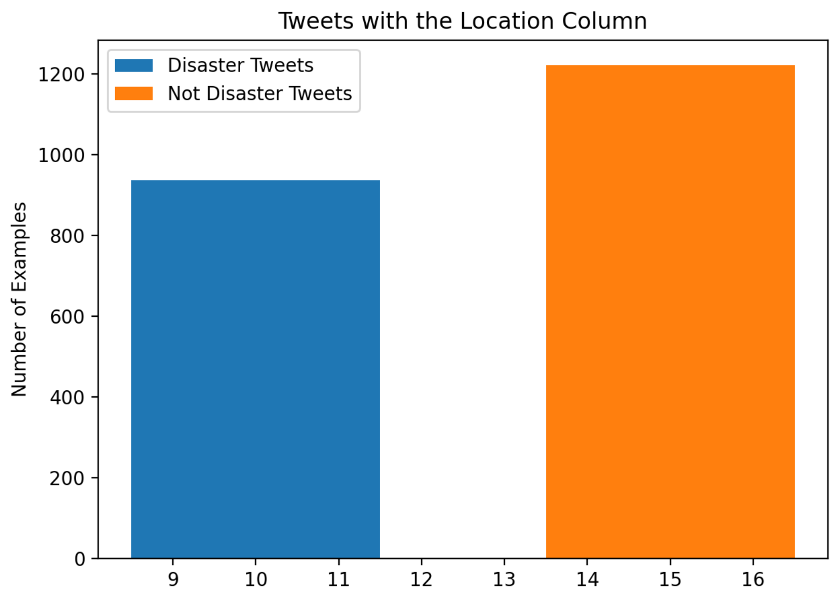


Figure 1.6: Location Column Distribution Chart for the Test Data Set



It can be seen that disaster tweets and non disaster tweets have quite similar characters distribution. But, its fair to say that the non disaster tweets have more characters in general then the disaster tweets, as suggested in figures 1.7 and 1.8.

Figure 1.7: Characters Distribution Chart for the Train Data Set

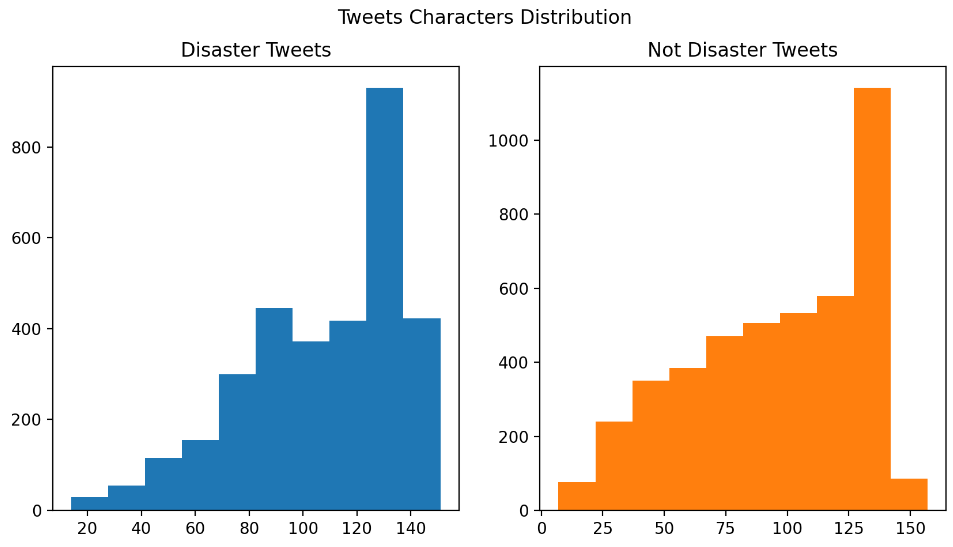
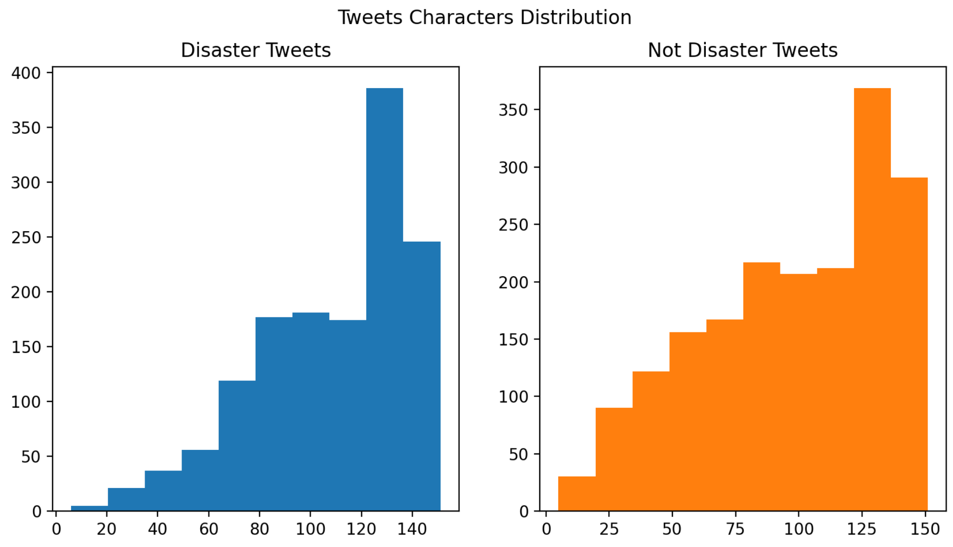


Figure 1.8: Characters Distribution Chart for the Test Data Set



Words number in tweets figures 1.9 and 1.10 also have similar shapes. Moreover, the scale for charts is the same. It is normal that non disaster tweets have a little bit bigger bars because there are more tweets of this class.

Figure 1.9: Tweets Words Number Chart for the Train Data Set

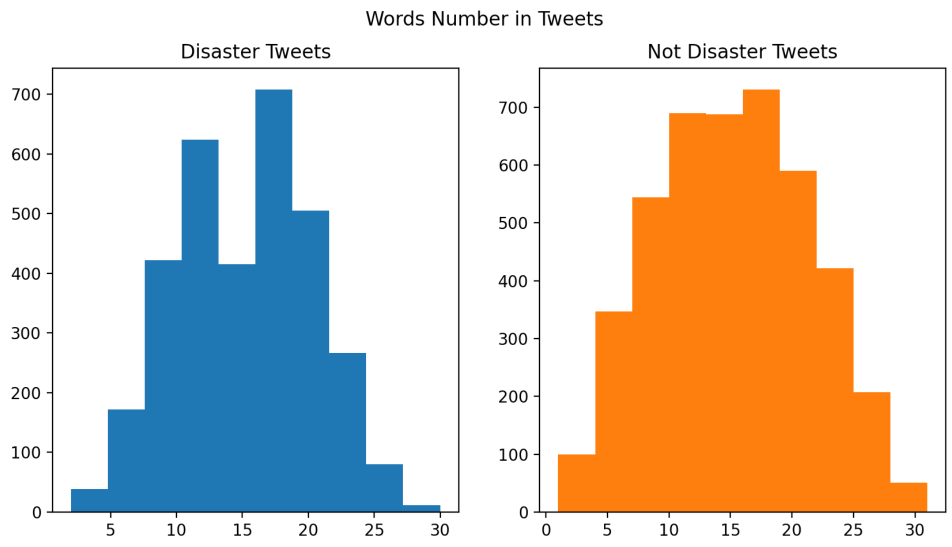
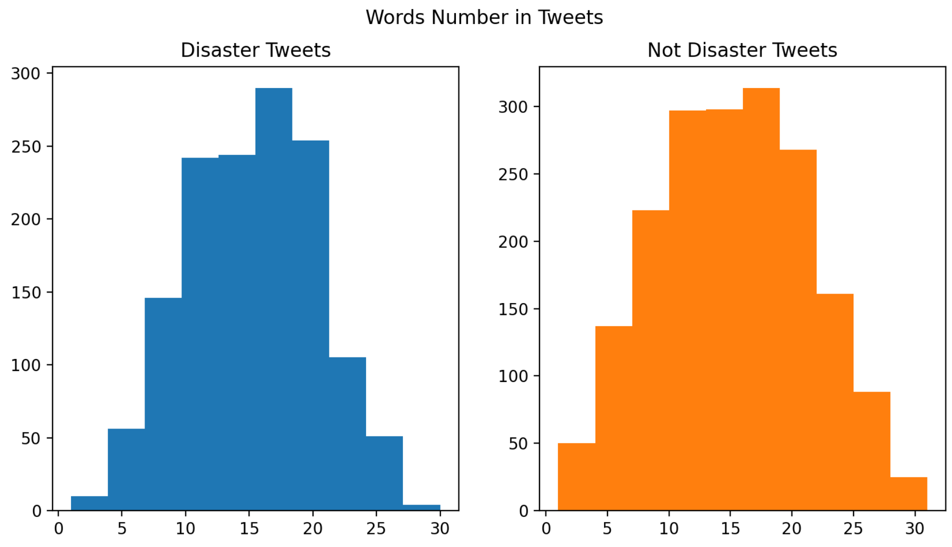


Figure 1.10: Tweets Words Number Chart for the Test Data Set



The same similarities in trends hold for the average word length figures. Train Data Set is shown in figure 1.11 and Test data set is shown in figure 1.12.

Figure 1.11: Tweets Average Word Length Chart for the Train Data Set

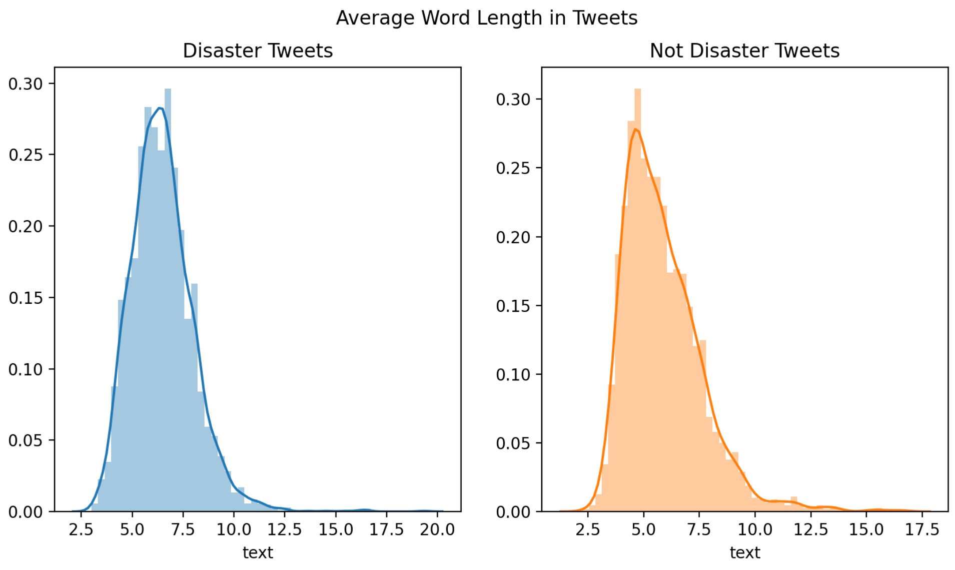
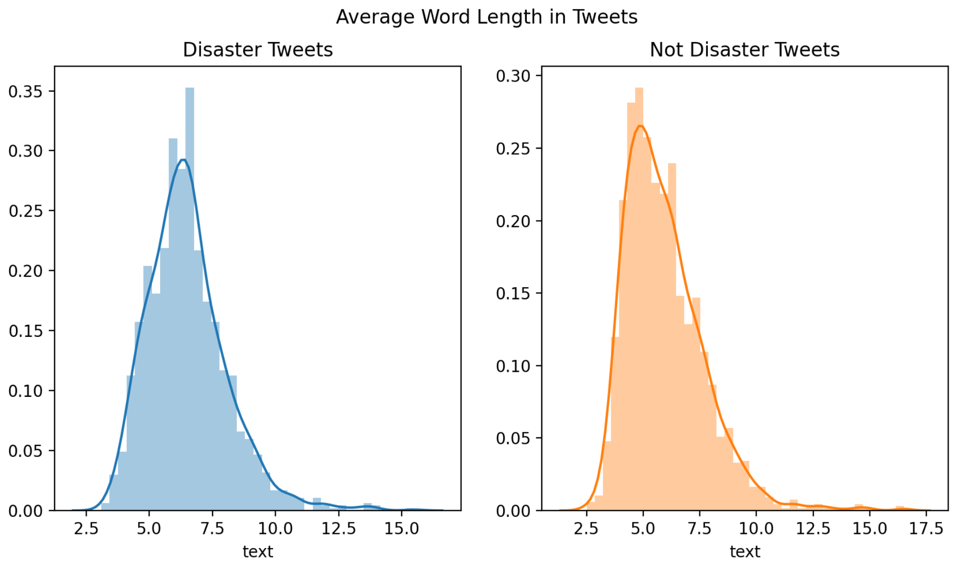


Figure 1.12: Tweets Average Word Length Chart for the Test Data Set



It can be seen that figures, which show top 10 stopwords, for the Train Data set 1.13 and for the Test Data Set 1.14 have almost same order of stopwords. However, there is a difference in sets of stopwords for each tweet classes. In both classes article *the* is the popular one. The higher number in non disaster tweets is normal, because there are more tweets of this class. In the non disaster class, there are more popular stopwords like: *i*, *you*, *my*. But, these stopwords are not present in disaster tweets class. In disaster tweets class, there are more "useful" stopwords: *in*, *of*, *on*, *for*, *at*. These are descriptive stopwords that can describe details of the disaster.

Therefore, stopwords should prove usefulness in the training process.

Figure 1.13: Top 10 Stop Words Chart for the Train Data Set

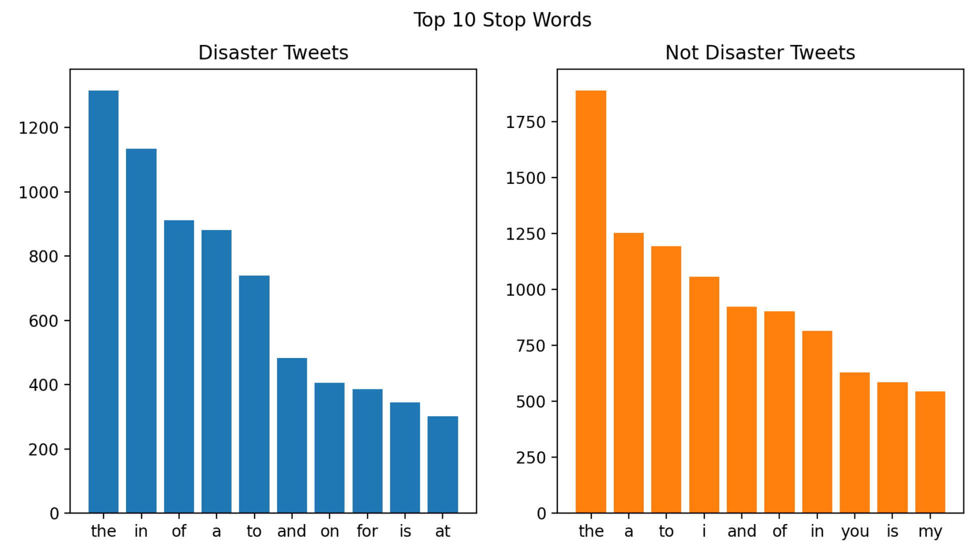
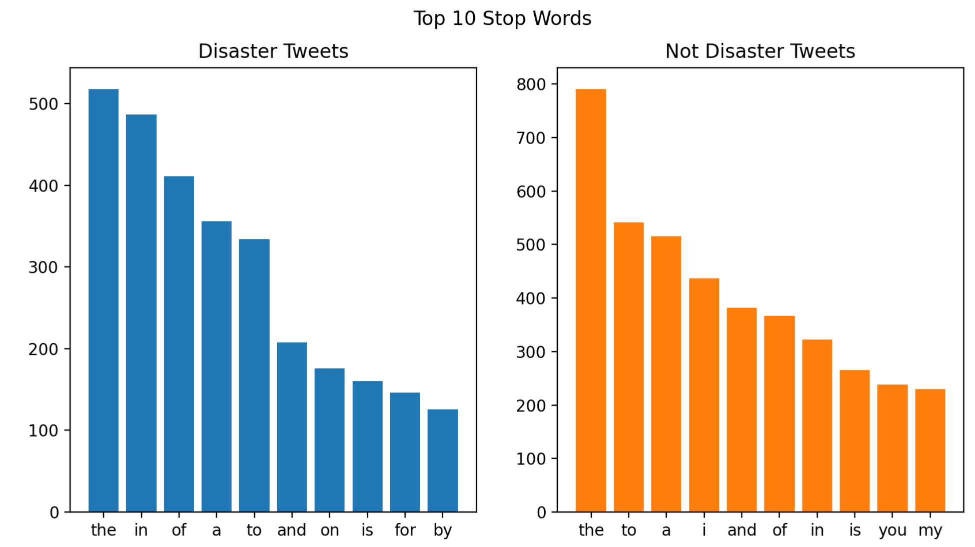


Figure 1.14: Top 10 Stop Words Chart for the Test Data Set



Punctuation figures for Train Data Set 1.15 and for Test Data Set 1.16 have a quite similar trends. Three charts have the same order of top 3 punctuation signs ('-', '|', ':'). It can be said that there is no visible correlation between punctuation and tweet class, because top 3 punctuation signs are present in both classes, in the same ratio.

Figure 1.15: Tweets Punctuation Chart for the Train Data Set

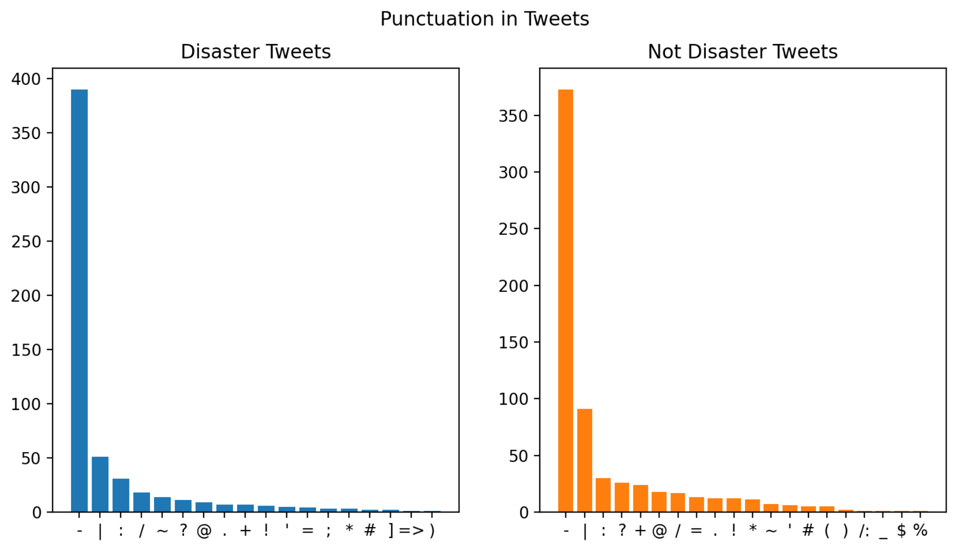
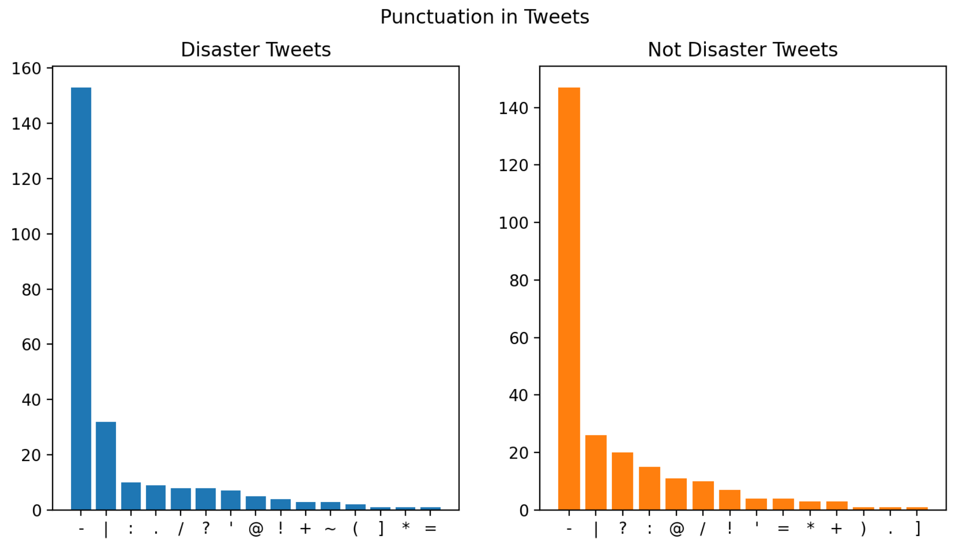


Figure 1.16: Tweets Punctuation Chart for the Test Data Set



In figures 1.17 and 1.18 top 10 hashtags for Train and Test data sets are presented. First feature, which can be noticed is that there is a fairly small amount of hashtags, in comparison to the number of tweets. For example, in the train data set the top hashtag *#news* in disaster tweets occurs less then 60 times. And in top not disaster tweet hashtag *#nowplaying* occurs around 20 times. Its fairly a small number in comparison to 7613 tweets from Train data set. There is a difference in context of hashtags between classes, but taking into account number of hashtags repetitions, training process cannot rely on hashtags themselves. The best idea would be to mark tweet as one, which includes hashtag, but hashtag itself should not be considered as self stand word (*#news* and *news* should NOT be presented as two different words). This is thus the approach we have applied in data preprocessing.

Figure 1.17: Top 10 Hashtags Chart for the Train Data Set

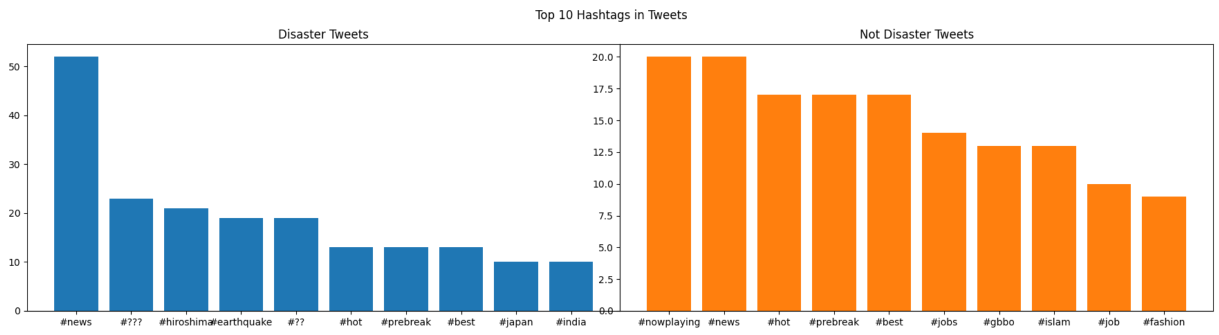
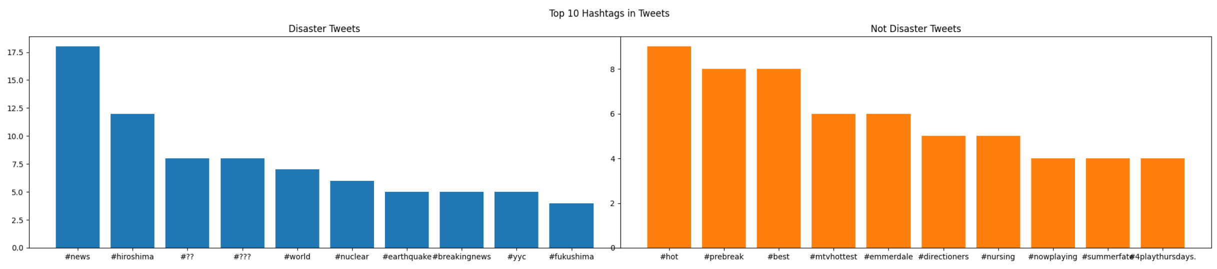


Figure 1.18: Top 10 Hashtags Chart for the Test Data Set



2. Programming Environment and Technical Aspects

In order to test a wide range of machine learning techniques several classes and large number of functions have been developed. All of them are aimed at creating an easily configurable environment, which with the minimum effort, lets one to test different known models and classifiers. This had to be done because it was decided to test overall six classifiers with different vectorizers and eleven neural network models with different embeddings on several preprocessing algorithms, which were described in the section 1.1.

2.1. Classes

The aim of creating an easily configurable programming environment was achieved with the *TweetsPreprocessor*, *Sklearn* and *Keras* classes. These classes are thoroughly examined in the 2.1.1, 2.1.2 and 2.1.3 subsections, respectively.

2.1.1. TweetsPreprocessor

Data preprocessing is an important step in the data mining process. It is a step where data can be transformed in a way to gain the best performance of the models. Data tends to have noise (not useful pieces of information), which can be filtered out, wheres at the same time, this data can be enriched with some extra information. For this purpose, there are already different open source libraries available which implement different preprocessing steps. NLTK [16] is a popular library, which specializes on the work with the natural language. Unfortunately, there is no a perfect library, which satisfies all the requirements. Thus, the class *TweetsPreprocessor*, responsible for data preprocessing, has been created. Partial code of this class can be examined in Appendix A.1. This class uses NLTK library, as a base for tweets tokenization, stop words recognition, etc. In order to use *TweetsPreprocessor* class, its instance should be created as in the figure 2.1.

Figure 2.1: TweetsPreprocessor instance creation example

```
tweets_preprocessor = TweetsPreprocessor(
    tokenizer=tweet_tokenizer,
    stemmer=porter_stemmer,
    stop_words=stop_words,
    slang_abbreviations=slang_abbreviations,
    splitters=splitters
)
```

Arguments' meaning list for the *TweetsPreprocessor* constructor is as follows:

- *tweet_tokenizer* is *nltk.TweetTokenizer* ([18]) instance.
- *porter_stemmer* is *nlt.stem.porter.PorterStemmer* ([17]) instance.
- *stopwords* is a union set of stopwords from *nltk.corpus* and *wordcloud* library
- *slang_abbreviations* is a dictionary {"afk": "away from keyboard", ...}
- *splitters* is a dictionary of symbols {"'", '-', ':', ' ', '_', '/'}. If word contains any of these symbols it is split.

Instance of this class has a method called *preprocess*, which takes a preprocessing algorithm dictionary as one of parameters. This dictionary defines how tweets data will be preprocessed. In figure 2.2, it is shown, how this method can be used.

Figure 2.2: Tweets Preprocessing

```
PREPROCESSING_ALGORITHM = {
    'add_link_flag': True,
    'add_user_flag': True,
    'add_hash_flag': True,
    'add_number_flag': True,
    'add_keyword_flag': True,
    'add_location_flag': True,
    'remove_links': True,
    'remove_users': True,
    'remove_hash': True,
    'unslang': True,
    'split_words': True,
    'stem': True,
    'remove_punctuations': True,
    'remove_numbers': True,
    'to_lower_case': True,
    'remove_stop_words': True,
    'remove_not_alpha': True,
    'join': True
}

data = pd.read_csv('train.csv')

tweets_preprocessor.preprocess(
    data['text'],
    PREPROCESSING_ALGORITHM,
    data['keywords'],
    data['locations']
)
```

Arguments' meaning list for the *preprocess* method of the *TweetsPreprocessor* instance, are as follows:

- *tweets* – is a *text* column from the parsed csv file
- *options* – is a preprocessing algorithm dictionary
- *keywords* – is a *keyword* column from the parsed csv file
- *locations* – is a *location* column from the parsed csv file

Each of these preprocessing algorithms parameters is set to *True* by default. So if to pass such parameters as in figure 2.3, it would mean that every parameter is set to *True* except *stem*.

Figure 2.3: Preprocessing Algorithm Default Options

```
PREPROCESSING_ALGORITHM = {
    'stem': False,
}

data = pd.read_csv('train.csv')

tweets_preprocessor.preprocess(
    data['text'],
    PREPROCESSING_ALGORITHM,
    data['keywords'],
    data['locations']
)
```

Meaning of preprocessing algorithm parameters is as follows:

- *add_link_flag* adds ' <url>' flag to the tweet ('I am an url https://www.google.com' -> 'I am an url https://www.google.com <url>')
- *add_user_flag* adds ' <user>' flag to the tweet ('I am @maryan' -> 'I am @maryan <user>')
- *add_hash_flag* adds ' <hashtag>' flag to the tweet ('I #am a #hashtag' -> 'I #am a #hashtag <hashtag>')
- *add_number_flag* adds ' <number>' flag to the tweet ('It's 2pm' -> 'It's 2pm <number>')
- *add_keyword_flag* adds ' <keyword>' flag to the tweet based on the column *keyword*

- *add_location_flag* adds ' <location>' flag to the tweet based on the column *location*
- *remove_links* removes urls from the tweet tokens list (['I', 'am', 'an', 'url', 'https://www.google.com', '<url>'] -> ['I', 'am', 'an', 'url', '<url>'])
- *remove_users* removes users from the tweet tokens list (['I', 'am', '@maryan', '<user>'] -> ['I', 'am', '<user>'])
- *remove_hash* removes hash from the tweet tokens list (['I', '#am', 'a', '#hash#tag', '<hashtag>'] -> ['I', 'am', 'a', 'hashtag', '<hashtag>'])
- *unslang* unslangs the tweet tokens list using slang_abbreviations passed to the class constructor (['afaik', 'it', 'will', 'be', 'today'] -> ['as', 'far', 'as', 'i', 'know', 'it', 'will', 'be', 'today'])
- *split_words* splits words in the tweet tokens list using splitters passed to the class constructor (['1-2:3 4/5'6'] -> ['1', '2', '3', '4', '5', '6'])
- *stem* stems words in the tweet tokens list (['stemming'] -> ['stem'])
- *remove_punctuations* removes all tokens which contains only punctuation symbols (['!!?', 'word!', '...'] -> ['word!'])
- *remove_numbers* removes any token containing number (['hello1', '2pm', '13 000', '.2'] -> [])
- *to_lower_case* every token is lowered cased
- *remove_stop_words* removes all stopwords from the tweet tokens list using stopwords passed to the class constructor
- *remove_not_alpha* every token which is not a number or a hashtag or a flag and contains not alphabetical symbol is removed
- *join* joins back list of tokens to single string tweet

PREPROCESSING_ALGORITHM dictionary can be used to configure the preprocessing algorithm. This class is thoroughly covered with unit tests, in order to be sure that if it was decided to keep hashtags they will not be removed by some other method like *remove_not_alpha*.

There are options: *add_link_flag*, *add_user_flag*, *add_hash_flag*, *add_number_flag*, *add_keyword_flag*, *add_location_flag* that add flags to the string. These flags were chosen because the content of these items might not be important. For example the link string itself, in the tweet, has no useful information. But the fact that the tweet has a link or user mention, or hashtag, or numbers, or location might be crucial. GloVe (Global Vectors for rd Representation) [19] have similar kinds of flags included. It has vector representations for *Link*, *User*, *Hash* and *Number* flags. Taking into account the information gathered from data analysis chapter 1.2.2, it is assumed that *keyword* and *location* column will be not useful. Nevertheless, these flags will be included for the testing purposes.

2.1.2. Sklearn

The Sklearn class is a scikit-learn [25] library wrapper class, which extracts common code and imports it into one place. The partial code of this class can be seen in the Appendix A.2.

Sklearn class is built of:

- *VECTORIZERS* - static dictionary which stores *Tfidf* [27] and *Count* [24] Sklearn vectorizers

- *CLASSIFIERS* - static dictionary which stores *Ridge*, *Logistic Regression*, *Random Forest*, *Decision Tree*, *SVM* and *SGD* classifiers from the scikit-learn library, which were mentioned in the state-of-the-art section 1.1.

This class enables easy utilization of the scikit-learn classifiers, in the Python program, without importing them directly from the library. In the figure 2.4 its shown how this class is intended to be used. By doing so, there is no need to create several programs for testing each classifier. It is enough to change *CLASSIFIER* and *VECTORIZER* variables to the desired ones.

Figure 2.4: Sklean Class use example

```
from models.sklearn import Sklearn

CLASSIFIER = {
    'TYPE': 'LOGISTIC_REGRESSION',
    'OPTIONS': {}
}

VECTORIZER = {
    'TYPE': 'TFIDF',
    'OPTIONS': {
        'binary': True,
        'ngram_range': (1, 1)
    }
}

vectorizer = Sklearn.VECTORIZERS[VECTORIZER['TYPE']](
    **VECTORIZER['OPTIONS']
)

classifier = Sklearn.CLASSIFIERS[CLASSIFIER['TYPE']](
    **CLASSIFIER['OPTIONS']
)
```

2.1.3. Keras

In this work Keras [12] (Python deep learning library) was used to implement ten neural networks (NN) models and, additionaly, the BERT model. All of them have been described at the end of section 1.1. Keras is a high-level neural networks API, written in Python. For the purpose of our disaster detection problem, there was a Keras wrapper class created. As all of the NN models, which were implemented, are solving a binary classification problem, thus there is a common code, which was generalized and extracted for each model into this Keras wrapper class. Partial code of this class can be found in the Appendix A.3. Explanation of each property and method is described further in this section.

Implemented Keras class is a wrapper class that extracts common functionality into one place and creates an easier interface to test models:

- *OPTIMIZRERS* - static dictionary which stores *Adam* and *RMSprop* optimizers from Keras Library.
- *DEFAULTS* - static dictionary which stores default model configuration values.
- *get_sequential_model* - is one of the main static methods that returns an instance of Keras.Sequential [13] model for the binary classification problem. As arguments it takes *layers*, *config*. Where *layers* is a list of Keras Layers which should be added to the model and where *config* is a dictionary with model configuration parameters. As the last layer is always the same (*Dense(1, activation="sigmoid")*), argument *layers* should be a list of Keras layers without the last one. To use Keras Embedding Layer *embedding_options* should be passed and the function will handle this layer by itself. More information about the Keras library can be found in [12].
- *_get_lstm_model* - is a private static method which returns Keras.Sequential model instance with the configured LSTM layer. As an argument this method takes *config* dictionary which stores the configuration parameteres of the model. One of such parameters is *LSTM_UNNITS*, which stores the number of LSTM Layer units.
- *_get_lstm_dropout_model*, *_get_bi_lstm_model*, *_get_fast_text_model*, *_get_rcnn_model*, *_get_cnn_model*, *_get_rnn_model* and *_get_gru_model* are similar static private methods to *_get_lstm_model*. All of them return configured Keras Sequential models.
- *get_model* - is a public static method which returns configured Keras.Sequential model. As a parameter it takes model's configuration dictionary and uses it's *TYPE* key in order to use the right private method for model configuration and creation.
- *get_bert_model* - returns BERT model instance. Implementation of BERT model was inspired by [23].
- *_plot* and *draw_graph* - are methods which are responsible for drawing two charts out of model training history. The first chart shows accuracy lines, wheres the second chart displays loss lines.
- *fit* - is a public static method, which is a wrapper for Keras *model.fit* method. It takes as arguments model instance, data tuple which contains train, validation and test splits,

and as a last argument it takes model config dictionary. Apart from calling *fit* method on the model instance, this method handles additional callbacks and returns history object dictionary.

Let us consider the *TestDataCallback*. Instance of this class can be passed to the model's *fit* method, as one of the arguments. In the current case, instance of this class has a callback function, which will be executed at the end of each epoch. This callback function is needed to evaluate how the model performs on the Test Data Set. Train and Validation accuracies are calculated and saved automatically, by Keras, to the history dictionary. As mentioned before, in the Data Set section 1.2, the Test data set is also used to check the performance of the models. So in the end of the model training, accuracy and loss histories of training, validation and test data sets will be saved.

2.2. Programs

There were two groups of Python programs created: a kfold programs group, and a single model programs group. These programs utilize classes from the sections 2.1.1, 2.1.2 and 2.1.3. These classes provide a possibility to test multiple models and classifiers on multiple preprocessing algorithms in a single program, by defining configuration dictionaries for preprocessing algorithm, model or classifiers. So there is no need to create separate program for each model and classifier type.

In order to test the performance of models and classifiers, *app_kfold_sklearn.py*, *app_kfold_keras.py* and *app_kfold_bert.py* (kfold programs group) programs were implemented. Each of these programs implements the StratifiedKFold [26] cross-validation. Every executed program saves all logs from the run experiments into dedicated log files. These logs files enable possibility for further analysis, like finding the averages and the highest results.

Along with the mentioned programs, there were also created *app_sklearn.py*, *app_keras.py* and *app_bert.py* (single programs group). These programs' aim was to run a single model, or classifier, with a particular configuration and with the selected preprocessing algorithm, in order to save a trained model, or classifier, to the file for the future use. Saved models were used for generating predictions vectors, which were used for further training.

In the following sections above mentioned programs are described in more details.

Additionally, there were other helper programs created, which were responsible for model and classifier logs analysis, csv reports generating, data sets analysis, etc. They will not be described in this work as they are not relevant. Please note that the symbol "...", in the Appendix codes, means that some parts of the code were omitted.

2.2.1. Sklearn

Let us consider *app_kfold_sklearn.py* from the Appendix B.1. In this program there are two lists *VECTORIZERS* and *CLASSIFIERS*. *VECTORIZERS* list contains constructor configurations for *Tfidf* and *Count* vectorizers. Each of these vectorizers has different *ngram_range* values. In this work there were (1,1), (1,2) and (1,3) *ngrammars* tested. *CLASSIFIERS* list stores six classifiers types (*RIDGE*, *LOGISTIC_REGRESSION*, *RANDOM_FOREST*, *DECISION_TREE*, *SVC* and *SGD*), which were decided to be tested, in section 1.1. Later, there are four main cycles, with help of which each classifier is tested on each vectorizer and on each preprocessing algorithm, using 10-fold cross-validation. All of the results are logged into a log file.

The *app_sklearn.py* program is quite similar to the *app_kfold_sklearn.py*. The differences are that there is only one classifier, one vectorizer and one preprocessing algorithm selected. Also there is no 10-fold cross-validation implemented. This program saves the trained classifier into the file which can be loaded and used later.

2.2.2. Keras

Let us consider *app_kfold_keras.py* from the Appendix B.2. This program can be executed in two modes: with use of GloVe embeddings and without. The *USE_GLOVE* variable decides on the mode. If its value is set to *True* then GloVe embeddings will be loaded and used. *get_glove_embeddings* function loads this embeddings from the file and stores them in *GLOVE_EMBEDDINGS* variable. *NETWORKS_KEYS* is a list of models which are going to be tested. Thus, for each model key, the *get_model_config* function is called, which returns base predefined model configuration dictionary. Models configuration dictionaries are stored in a different file. In the figure 2.5 there is an example of the base configuration dictionary for the LSTM model.

Figure 2.5: LSTM model configuration dictionary example

```
{
    'TYPE': 'LSTM',
    'BATCH_SIZE': 32,
    'EPOCHS': 20,
    'OPTIMIZER': 'adam',
    'LEARNING_RATE': 1e-4,
    'EMBEDDING_OPTIONS': {
        'output_dim': 200,
    },
    'LSTM_UNITS': 100,
}
```

The base model configuration dictionary is stored in the *MODEL_CONFIG* variable. This dictionary is a template, which will be copied into *CONFIG* variable at each preprocessing algorithm cycle step. This is done to omit base dictionary mutations, and to persist unique values for logs. *CONFIG* dictionary is populated, at each step, with information about preprocessing algorithm, its *id* and *KFOLD_HISTORY*. At each cycle, step log file is updated with the cycle step calculations.

The *app_keras.py* program is quite similar to the *app_kfold_keras.py*. But it does all of the above only for one model and one preprocesising algorithm. The output of this programs is a saved model, which can be loaded and used in the future.

2.2.3. Bert

BERT is a pretrained model and it uses a different embedding mechanism than others neural network models. Due to this fact, a separate program was required. BERT model itself, and its requirements implementation, was inspired by [23]. Let us consider *app_kfold_bert.py*, from Appendix B.3. BERT is loaded using the tfhub library – TensorFlow hub [8]. The programs' code is quite similar to *app_kfold_keras.py* (Appendix B.2), as both of them are using the same Keras library, and the same library wrapper class. But this program handles model creation and its embeddings in a different way. The output of this program is a log file.

app_bert.py program is almost the same as the *app_kfold_bert.py*, except it trains only one BERT model with the selected preprocessing algorithm, and does not use kfold cross-validation.

BERT model file is created for the future use as an output of the executed program.

3. Results

In this chapter the achieved results and conducted experiments will be described. The aim of this work was to apply, broadly understood, machine learning tools and methods, to analyse content of tweets and to detect disasters. Therefore, there were selected six classifiers (Logistic Regression, Ridge, SVC, SGD, Decision Tree and Random Forest) and eleven neural networks (NN) models (LSTM, Bi-LSTM, LSTM-Dropout, LSTM-CNN, FastText, RNN, RCNN, CNN, GRU, HAN and BERT), as it was mentioned in the state-of-the-art section 1.1. All these models and classifiers were tested on ten different data preprocessing algorithms, which are described in section 3.1. It was done to find a universal data preprocessing algorithm, which would succeed with all models, or check the theory that each model would perform best with a different data preprocessing algorithm.

Apart from testing selected models and classifiers on ten different data preprocessing algorithms, they were tested with different configurations. Six classifiers were tested with the Count and Tfidf vectorizers with the different ngrammar ranges. Their averages and the highest results are presented in section 3.2. Ten NN models (excluding the BERT model) were tested with the usage of Keras Embeddings. Results of those experiments are summarized in the section 3.3. These ten NN models were also tested with the usage of GloVe Embeddings. Their results are described in section 3.4. The BERT model experiments and results are shown in section 3.5. They are described in a separate section because BERT is a pre-trained model, which required an embedding mechanism different from others ten NN models. To evaluate this model, there was a separate program developed, which was described in the section 2.2.3. All of experiments in these sections were executed using StratifiedKFold [26] with 10 splits and seed equal to 7. Full results for all models and classifiers can be found in the google sheet document [22]. This is provided since the actual volume of generated data is such, that it cannot be properly depicted in the Thesis itself.

During the experiments, described in sections 3.2, 3.3, 3.4 and 3.5 each model and classifier performance was evaluated, for each preprocessing algorithm. Based on the highest average results each model and classifier was paired with its best preprocessing algorithm.

The first result obtained during our work was that we were able to establish 27 pairs (preprocessor+model) that were further investigated as having best potential for delivering quality results. Specifically, we have selected six classifiers, ten models base on Keras embeddings, ten models based on GloVe embeddings, BERT model, combined with their best preprocessing algorithms.

Afterwards, the Test data set was analysed using predictions of these best 27 pairs. Its summary is presented in the section 3.7.

Moreover, based on the predictions generated by the 27 pairs, a new, simple, NNs were trained to gain a better result. This process is thoroughly described in section 3.6.

Additionally, simple weights formulas were applied to calculate a final predictions list using predictions of 27 best pairs. Their results are presented in 3.8.

Finally, it was decided to analyse the Train data set and the Train+Test data set, and discover if removing the group of "bad tweets" from the training data set (tweets, for which all of 27 pairs has failed to recognize a correct answer), would produce a better score. Here, we have decided that while the Kaggle competition setup should be applied "as is" (Train data set), we can also use the complete, tagged, data set (Train+test data set), to explore its characteristics vis-a-vis tweet classification. These experiments and results are described in the section 3.9 and 3.10 for the Train data set and for the Train+Test data set respectively. Full tables with the results for these two sections can be found in the google sheets [20] and [21].

3.1. Selected Preprocessing Algorithms

As mentioned in section 2.1.1, data preprocessing is a very important step. It helps to prepare data for machine learning algorithms by removing noise and enriching it with useful (explicitly and implicitly available) information. It is a well known fact that good data preparation can significantly increase the accuracy of the final results [5].

There are seventeen configurable parameters in the *TweetPreprocessing.preprocess* method, which were described in chapter 2.1.1. Each parameter can be either True or False. This means that 2^{17} different algorithms can be created. It is too time consuming to check the performance of each algorithm, especially when it is known that the majority of these algorithms will perform poorly. Thus, there were ten preprocessing algorithms selected. Each of them has same "frozen" parameters. In each algorithm, the frozen parameters are: *Stem - False*, *Lower - True*, *Unslang - True*, *Remove Links - True*, *Remove Users - True*, *Remove Hash - True*, *Remove Numbers - True*. These frozen parameters were selected experimentally. Specifically, changing some of them to opposite boolean value had decreased the algorithm performance severely. These 10 algorithms are presented in the tables 3.1, 3.2 and 3.3. Each algorithm has a unique *ID*, by which it will be referred in the results tables.

Table 3.1: Preprocessing Algorithms Flags

ID	*Link Flag*	*User Flag*	*Hash Flag*	*Number Flag*	*Keyword Flag*	*Location Flag*
1258a9d2	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
60314ef9	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE
4c2e484d	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
8b7db91c	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
7bc816a1	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
a85c8435	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
b054e509	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
2e359f0b	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
71bd09db	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
d3cc3c6e	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE

Table 3.2: Preprocessing Algorithms Removes

ID	*R Links*	*R Users*	*R Hash*	*R Punct*	*R Number*	*R StopWords*	*R NotAlpha*
1258a9d2	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
60314ef9	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
4c2e484d	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
8b7db91c	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
7bc816a1	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
a85c8435	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE
b054e509	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE
2e359f0b	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE
71bd09db	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
d3cc3c6e	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE

Table 3.3: Preprocessing Algorithms rest manipulations

ID	*Unslang*	*Split Words*	*Stem*	*Lower*
1258a9d2	TRUE	TRUE	FALSE	TRUE
60314ef9	TRUE	TRUE	FALSE	TRUE
4c2e484d	TRUE	TRUE	FALSE	TRUE
8b7db91c	TRUE	TRUE	FALSE	TRUE
7bc816a1	TRUE	TRUE	FALSE	TRUE
a85c8435	TRUE	TRUE	FALSE	TRUE
b054e509	TRUE	FALSE	FALSE	TRUE
2e359f0b	TRUE	FALSE	FALSE	TRUE
71bd09db	TRUE	FALSE	FALSE	TRUE
d3cc3c6e	TRUE	FALSE	FALSE	TRUE

Seventeen algorithm parameters were divided into three groups, and are presented in three tables. Table 3.1 contains "flag presence parameters" group, which configures the presence of *Link*, *User*, *Hash*, *Number*, *Keyword* and *Location* flags in the tweet. Table 3.2 contains "remove parameters" group, which removes some information from the tweet (*R Links*, *R Users*, *R Hash*, *R Punct*, *R Number*, *R StopWords* and *R NotAlpha*). And the 3.3 table contains the rest of parameters (*Unslang*, *Split Words*, *Stem* and *Lower*).

First five algorithms, 1258a9d2, 60314ef9, 4c2e484d, 8b7db91c and 7bc816a1, were selected to test how the presence of *Link*, *User*, *Hash*, *Number*, *Keyword* and *Location* flags, in the preprocessed tweet, influences the algorithm performance. Their parameters' differences can be checked in the 3.1 table. All other non frozen parameters are set to *True* and are the same for these five algorithms (3.2, 3.3).

Other three algorithms, a85c8435, b054e509 and d3cc3c6e, plus one algorithm 1258a9d2 from the first five, have similar configuration for the "flag presence parameters" group (3.1), but different configurations for other parameters groups. The four parameters from "flag presence parameters" (*Link Flag*, *User Flag*, *Hash Flag* and *Number Flag*) are set to *True* because they have vector representations in Global Vectors for Word Representation (GloVe). These four algorithms test how *Split Word*, *Remove Punctuation*, *Remove Stop Words*, and *Remove Not Alpha* parameters influence the performance of the algorithm.

The other two algorithms, 2e359f0b and 71bd09db, plus 7bc816a1 from the first five, have the same configuration values, all set to *False*, for the "flag presence parameters" group in table

3.1. These three algorithms test how *Split Word*, *Remove Punctuation*, *Remove Stop Words*, and *Remove Not Alpha* parameters influence the performance of algorithm.

3.2. Sklearn Classifiers

In this part of our work, we tested Ridge, Linear Regression, Logistic Regression, Random Forest, Decision Tree, SVC and SGD classifiers. These classifiers were selected in section 1.1. Each of them was tested with ten different preprocessing algorithms using *Tfidf* and *Count* vectorizers, with different grammar ranges ([1, 1], [1, 2], [1, 3]). Each experiment was executed using 10-fold cross-validation approach. Tables in this chapter present the average test scores of ten folds.

Table 3.4: Top 15 classifiers experiments

Classifier	Vectorizer [grammar range]	Algorithm ID	Score
Ridge	Tf-idf [1, 2]	2e359f0b	0.8044439
Ridge	Tf-idf [1, 2]	71bd09db	0.8044439
SVC	Count [1, 1]	4c2e484d	0.8025131
Logistic Regression	Count [1, 2]	8b7db91c	0.8024518
SVC	Count [1, 1]	8b7db91c	0.8023291
Ridge	Tf-idf [1, 3]	60314ef9	0.8021147
Logistic Regression	Count [1, 3]	8b7db91c	0.8021145
Ridge	Tf-idf [1, 3]	71bd09db	0.8019614
Ridge	Tf-idf [1, 3]	2e359f0b	0.8019614
SVC	Tf-idf [1, 1]	d3cc3c6e	0.8016549
SVC	Tf-idf [1, 1]	b054e509	0.8016243
SGD	Tf-idf [1, 3]	2e359f0b	0.8015629
Ridge	Tf-idf [1, 2]	60314ef9	0.8015323
Logistic Regression	Tf-idf [1, 2]	60314ef9	0.8015323
SVC	Tf-idf [1, 1]	a85c8435	0.8014710

There are top fifteen experiments, with the best results on the Test data set, presented in table 3.4.

Table 3.5: Top 3 Ridge classifiers experiments based on the best experiment parameters

Classifier	Vectorizer [grammar range]	Algorithm ID	Score
Ridge	Tf-idf [1, 2]	2e359f0b	0.8044439
Ridge	Tf-idf [1, 2]	71bd09db	0.8044439
Ridge	Tf-idf [1, 2]	60314ef9	0.8015323

Top three results for the Ridge Classifier, based on the best experiment parameters (vectorizer and grammar range), are presented in 3.5.

Table 3.6: Top 3 Logistic Regression classifiers experiments based on the best experiment parameters

Classifier	Vectorizer [grammar range]	Algorithm ID	Score
Logistic Regression	Count [1, 2]	8b7db91c	0.8024518
Logistic Regression	Count [1, 2]	1258a9d2	0.8014404
Logistic Regression	Count [1, 2]	4c2e484d	0.8011952

Top three results, for the Logistic Regression Classifier, based on the best experiment parameters (grammar range, vectorizer) are presented in table 3.6.

Table 3.7: Top 3 SVC experiments based on the best experiment parameters

Classifier	Vectorizer [grammar range]	Algorithm ID	Score
SVC	Count [1, 1]	4c2e484d	0.8025131
SVC	Count [1, 1]	8b7db91c	0.8023291
SVC	Count [1, 1]	1258a9d2	0.8008274

Top three results, for the SVC, based on the best experiment parameters (grammar range, vectorizer) are presented in table 3.7.

Table 3.8: Top 3 SGD classifier experiments based on the best experiment parameters

Classifier	Vectorizer [grammar range]	Algorithm ID	Score
SGD	Tf-idf [1, 3]	2e359f0b	0.8015629
SGD	Tf-idf [1, 3]	60314ef9	0.8010726
SGD	Tf-idf [1, 3]	71bd09db	0.8004903

Top three results, for the SGD classifier, based on the best experiment parameters (grammar range, vectorizer) are presented in table 3.8.

Table 3.9: Top 3 Random Forest classifier experiments based on the best experiment parameters

Classifier	Vectorizer [grammar range]	Algorithm ID	Score
Random Forest	Tf-idf [1, 1]	60314ef9	0.7888141
Random Forest	Tf-idf [1, 1]	8b7db91c	0.7865767
Random Forest	Tf-idf [1, 1]	7bc816a1	0.7833282

Top three results for, the Random Forest classifier, based on the best experiment parameters (grammar range, vectorizer) are presented in table 3.9.

Table 3.10: Top 3 Decision Tree classifier experiments based on the best experiment parameters

Classifier	Vectorizer [grammar range]	Algorithm ID	Score
Decision Tree	Count [1, 2]	7bc816a1	0.7633159
Decision Tree	Count [1, 2]	60314ef9	0.7592093
Decision Tree	Count [1, 2]	8b7db91c	0.7464910

Top three results, for the Decision Tree classifier, based on the best experiment parameters (grammar range, vectorizer) are presented in table 3.10.

Comparing results presented thus far, it cannot be said that a specific preprocessing algorithm is a definite champion among others. Calculated relative scores of preprocessing algorithms, are presented in table 3.11. They are calculated based on the algorithm position in tables 3.5, 3.6, 3.7, 3.8, 3.9 and 3.10. If algorithm is on the first place it receives 3 points, for the second place 2 points and for the last place -1 point.

Table 3.11: Preprocessing algorithms score based on tables above

Algorithm	Positions Points	Score
71bd09db	3 + 1 + 1 + 3	8
8b7db91c	3 + 2 + 2 + 1	8
60314ef9	1 + 2 + 3 + 2	8
2e359f0b	3 + 3	6
4c2e484d	1 + 3	4
1258a9d2	2 + 1	3

Using this scoring method (while keeping in mind its limitations) the best performance was achieved with **2e359f0b** and **71bd09db** preprocessing algorithm and the Ridge classifier. These two algorithms are almost the same, except for a single parameter. The **2e359f0b** algorithm does not remove punctuation from tweets, while **71bd09db** does. Both algorithms do not add any flags and do not remove stop words.

The second best was **8b7db91c**. It worked well with the Logistic Regression classifier. This preprocessing algorithm adds link, user, hash flags and removes punctuation, stop words. The third best is **60314ef9**, which is very similar to **8b7db91c**, except the **8b7db91c** adds link flag but **60314ef9** does not. Finally, the Decision Tree classifier and Random Forest classifiers performed the worst (regardless of the preprocessing).

In summary, we can conclude that there were three leaders among classifiers: Ridge, SVC and Logistic Regression. They performed best, but when paired with different preprocessing algorithms. Therefore there is no “absolute winner” among preprocessing algorithms, which can be said that it performs best with every classifier.

3.3. Keras Models based on Keras Embeddings

In this section we report work where we tested ten different neural network models: LSTM, LSTM-Dropout, Bi-LSTM, LSTM-CNN, Fast Text, RCNN, CNN, RNN, GRU and HAN. These networks were described in section 1.1. All these models are using the Embedding Layer, from the Keras library. All these models are tested with different hyper-parameters and, using ten different prepossessing algorithms. All the experiments are executed using the 10-fold approach. In table 3.12, there are top average scores for these ten neural network models. Each number in the table is an average score of 10-fold cross validation training.

Table 3.12: Top Averages Scores for 10 NN Models

Model	Optimizer	Algorithm ID	Score
LSTM	Adam	71bd09db	0.796843
LSTM_DROPOUT	RMSprop	1258a9d2	0.797824
BI_LSTM	Adam	8b7db91c	0.797456
LSTM_CNN	Adam	8b7db91c	0.793840
FASTTEXT	**Adam**	**b054e509**	**0.803739**
RCNN	RMSprop	d3cc3c6e	0.794177
CNN	Adam	b054e509	0.799939
RNN	Adam	1258a9d2	0.798590
GRU	RMSprop	a85c8435	0.800552
HAN	Adam	a85c8435	0.801716

It can be easily noticed that the average scores in table 3.12 are quite similar to each other.

In table 3.13, there are top highest results and their fold indexes, achieved by combinations of a neural network model and a preprocessing algorithm.

Table 3.13: Top Highest Scores for 10 NN Models

Model	Optimizer	Algorithm ID	Fold	Score
LSTM	Adam	b054e509	0	0.802636
LSTM_DROPOUT	Adam	71bd09db	8	0.805087
BI_LSTM	RMSprop	a85c8435	6	0.806313
LSTM_CNN	RMSprop	60314ef9	1	0.802329
FASTTEXT	RMSprop	a85c8435	8	0.806926
RCNN	RMSprop	71bd09db	7	0.804168
CNN	**RMSprop**	**2e359f0b**	**7**	**0.809378**
RNN	Adam	a85c8435	4	0.804781
GRU	RMSprop	71bd09db	4	0.807233
HAN	Adam	2e359f0b	4	0.805087

Overall, the same observation can be made for the NN models, based on Keras Embeddings, as for the Classifiers. There is no pair of preprocessing algorithm and NN model, which stands out significantly among others. However, the best average score was achieved with the b054e509 algorithm and the FastText NN model. Moreover, the highest score was achieved with the 2e359f0b algorithm and the CNN model on the 7th fold counting from 0.

3.4. Keras Models based on GloVe Embeddings

In this section we present results for the same set of NN models as in the previous one. The difference is that models in this case are using the GloVe embeddings. In table 3.14 there are top average scores for the ten neural network models with the GloVe embeddings.

Table 3.14: Top Averages Scores for 10 NN Models with GloVe

Model	Optimizer	Algorithm ID	Score
LSTM	Adam	8b7db91c	0.809960
LSTM_DROPOUT	RMSsprop	60314ef9	0.812565
BI_LSTM	Adam	8b7db91c	0.810389
LSTM_CNN	Adam	8b7db91c	0.806099
FASTTEXT	Adam	8b7db91c	0.812289
RCNN	Adam	71bd09db	0.804781
CNN	Adam	d3cc3c6e	0.806221
RNN	RMSprop	60314ef9	0.807876
GRU	**Adam**	**8b7db91c**	**0.814435**
HAN	Adam	a85c8435	0.809746

It can be seen that the averages in the table 3.14 are higher, in general than these in table 3.12. So, we can conclude that using GloVe embeddings increased the performance of our models. Also it can be seen that the 8b7db91c algorithm was at the top for five out of ten models. Moreover, the GRU model, in pair with the 8b7db91c algorithm, had the top average result.

Table 3.15: Top Highest Scores for 10 NN Models with GloVe

Model	Optimizer	Algorithm ID	Fold	Score
LSTM	RMSprop	8b7db91c	4	0.817652
LSTM_DROPOUT	RMSprop	60314ef9	8	0.821024
BI_LSTM	Adam	8b7db91c	7	0.819491
LSTM_CNN	Adam	2e359f0b	9	0.811523
FASTTEXT	RMSprop	8b7db91c	7	0.818265
RCNN	Adam	60314ef9	0	0.816120
CNN	Adam	7bc816a1	5	0.813362
RNN	Adam	8b7db91c	8	0.817346
GRU	**Adam**	**a85c8435**	**7**	**0.821943**
HAN	Adam	4c2e484d	4	0.816120

In table 3.15, there are top highest scores and their fold indexes, achieved for the NN models in pair with preprocessing algorithms.

The same observation holds for table 3.15. The 8b7db91c algorithm scored best in four out of ten models. But the highest result was achieved with the GRU model and the a85c8435 algorithm, on the 7th fold index counting from 0. It is safe to say that the 8b7db91c algorithm stands out among others with models, which are using GloVe embeddings.

3.5. BERT Model

In this section experiments with the BERT-Large model, from the TensorFlow Models repository on GitHub at tensorflow/models/official/nlp/bert [7]. It uses L=24 hidden layers (i.e. Transformer blocks), a hidden size of H=1024, and A=16 attention heads. This model is tested with ten preprocessing algorithm, and without the preprocessing phase. The average scores are presented in table 3.16.

Table 3.16: BERT Averages Scores for Preprocessing Algorithms

Model	Optimizer	Algorithm ID	Score
BERT	Adam	None	0.829911
BERT	Adam	1258a9d2	0.821085
BERT	Adam	60314ef9	0.822158
BERT	Adam	4c2e484d	0.821820
BERT	Adam	8b7db91c	0.821545
BERT	Adam	7bc816a1	0.816488
BERT	Adam	a85c8435	0.831535
BERT	Adam	b054e509	0.832516
BERT	**Adam**	**2e359f0b**	**0.834692**
BERT	Adam	71bd09db	0.828318
BERT	Adam	d3cc3c6e	0.825805

It can be noticed that the average scores with BERT model are much higher than average scores with other tested models and classifiers. The best average score is achieved in pair with the 2e359f0b preprocessing algorithm.

Table 3.17: Top Highest BERT

Model	Optimizer	Algorithm ID	Fold	Score
BERT	Adam	2e359f0b	5	0.840638

In table 3.17, there is a top highest score and its fold index, which was achieved with BERT model. This score was achieved with the 2e359f0b algorithm as well.

This score of 84,063% sets us at 79th place out of 3402 submissions in the Kaggle Leaderboard if do not include those scores which are better than 98%.

3.6. Combined Networks, NN Approach

The aim of this section is to discuss if the combination of predictions made by the best pairs of model/classifier and the best preprocessing algorithm could produce more accurate final predictions than a single best pair. In other words “if two heads are better than one”. In order to check this, a simple NN model was created and trained on the combined predictions data.

To generate predictions data set, it was decided to take the best performing pairs of model/classifier and preprocessing. Top six classifier pairs were taken from tables 3.5, 3.6, 3.7, 3.8, 3.9 and 3.10. Top ten pairs of NN models, based on the Keras embeddings, were taken from table 3.12, top ten pairs of NN models based on GloVe embeddings, from table 3.14, and top BERT pair from table 3.16. As a result 27 pairs of model/classifier and preprocessing were selected. They are presented in table 3.18.

Table 3.18: Top 27 Pairs of Model/Classifier and Preprocessing Algorithm

Model/Classifier	Preprocessing Algorithm ID
BERT	2e359f0b
LSTM-GloVe	8b7db91c
LSTM_DROPOUT-GloVe	8b7db91c
BI_LSTM-GloVe	8b7db91c
LSTM_CNN-GloVe	8b7db91c
FASTTEXT-GloVe	8b7db91c
RCNN-GloVe	71bd09db
CNN-GloVe	d3cc3c6e
RNN-GloVe	7bc816a1
GRU-GloVe	8b7db91c
HAN-GloVe	a85c8435
LSTM	71bd09db
LSTM_DROPOUT	d3cc3c6e
BI_LSTM	8b7db91c
LSTM_CNN	8b7db91c
FASTTEXT	b054e509
RCNN	7bc816a1
CNN	b054e509
RNN	1258a9d2
GRU	b054e509
HAN	a85c8435
RIDGE	2e359f0b
SVC	4c2e484d

Continuation of Table 3.18	
Model/Classifier	Preprocessing Algorithm ID
LOGISTIC_REGRESSION	8b7db91c
SGD	2e359f0b
DECISION_TREE	7bc816a1
RANDOM_FOREST	60314ef9

Each model/classifier and preprocessing pair from table 3.18 was trained on 70% of the (original Kaggle) Train data set, and validated against the remaining 30%. Please note that for each of them the same data split (the same data in both parts) was used. These trained models and classifiers produced 27 predictions lists for the validations sets. These predictions lists were transformed into input for the neural network:

$$[v_1, v_2 ..., v_n],$$

where v_i – is the list of 27 predictions per tweet; $i = 1..n$; n – number of tweets in the validation set (30% of Train data set). After training the neural network on this input (with the same split 70%/30%), the average results did barely beat the score of the standalone BERT pair.

In table 3.19, there are presented results for one of multiple experiments. These results are averages of 10-fold cross-validation approach. In this experiment, NNs were created using Keras library. They were built of Input Layer (IL), which size corresponded to the number of predictions sources used, and one Hidden Layer (HL), which was half the size of the Input Layer.

Table 3.19: Simple NN training summary

Pair Combination	Keras NN	Best Single Model Score	Average Score	Diff
B + 10GM + 10M + 6C	(IL: 27; HL: 14)	0.836040 (BERT)	0.836745	+0.000705
B + 10GM + 10M	(IL: 21; HL: 12)	0.836040 (BERT)	0.836623	+0.000583
B + 10GM	(IL: 11; HL: 6)	0.836040 (BERT)	0.833742	-0.002298
B + 10M	(IL: 11; HL: 6)	0.836040 (BERT)	0.834416	-0.001624
B + 6C	(IL: 7; HL: 4)	0.836040 (BERT)	0.834049	-0.001991
10GM + 10M + 6C	(IL: 26; HL: 13)	0.816120 (GRU-GloVe)	0.817898	+0.001778
10GM + 10M	(IL: 20; HL: 10)	0.816120 (GRU-GloVe)	0.818357	+0.002237
10GM + 6C	(IL: 16; HL: 8)	0.816120 (GRU-GloVe)	0.817530	+0.00141
10M + 6C	(IL: 16; HL: 8)	0.806926 (FastText)	0.801349	-0.005577
10GM	(IL: 10; HL: 5)	0.816120 (GRU-GloVe)	0.815814	-0.000306
10M	(IL: 10; HL: 5)	0.806926 (FastText)	0.802390	-0.004536
6C	(IL: 6; HL: 3)	0.801410 (Ridge)	0.803739	+0.002329

The maximum gain of such approach (in pairs combinations were BERT pair was used), on some folds, was no greater than 0.4%, in comparison to the single BERT pair result. Unfortunately, this approach did not overcome the result of the BERT pair from table 3.17. The BERT result from table 3.17 was achieved with 90%/10% train data split, on the 5th fold counting from 0.

It should be mentioned, that combination of predictions from the 10 pairs of NN models, based on GloVe embeddings, with 6 classifier pairs, on some folds of different NN applied, gave a gain around 1.5% in comparison to the best pair used in this combination. This indicates that combination of predictors would be useful if there was no big difference in individual scores. Here, the inputs from the model/classifier pairs, had score around 79-81%, and only one prediction input from BERT pair, which had score around 83.6%. Because of this, a considerable amount of bad predictions influenced the best BERT predictions. In other words NN could not find out that BERT is the most accurate. This seems to suggest that BERT should have the biggest impact (or, at least, “bigger than average” impact) on the final prediction.

3.7. Predictions Analysis of the Test Data Set

After failed attempts to beat the possible top Kaggle competition leaderboard score of 86.6%, it was decided to examine thoroughly pairs (of model/classifier and preprocessing algorithm from table 3.18) predictions for the Test data set. This was also caused by the lack of success of model combining. There, we have realized that we need to look in more detail what is actually happening during tweet classification. Specifically, we decided to look into questions like this: which tweets are "difficult"? are the same tweets difficult for all models? which tweets are "easy"? are the same tweets easy for all models? what is the tweet "coverage" by individual models?

In this context, in the Google sheet with the results [22], there were several tables created with 27 pairs' predictions on the (original Kaggle) Test data set. In these tables, 27 pairs can be compared against each other, based on predictions' values. These tables cannot be included in the work as they have more than 3000 rows and 30 columns.

The main table in [22] is called *TDS_27M*. It contains prediction values of mentioned 27 pairs for the Test data set. There were two additional tables created for the 27 pairs: *TDS_27M_FAILED* and *TDS_27M_SOME_FAILED*. Here, the *TDS_27M_FAILED* table contains only tweets, for which all 27 pairs (models) have failed to classify correctly. This means that for each of these tweets each model gave the wrong answer (yes for no or no for yes). The table *TDS_27M_SOME_FAILED*, on the other hand, contains set of tweets for which only some models have failed to identify particular tweet correctly, while some others did not. In other words, here, at least one model gave the correct answer. Obviously, the remaining tweets (after extracting tweets form these two tables) have been identified correctly (these are the "easy tweets"). Summary of these tables is presented in table 3.20.

Table 3.20: Summary of 27 pairs predictions

Table Name	Tweets Count	Percentage of Test Data Set
TDS_27M	3263	100%
TDS_27M_FAILED	159	4,87%
TDS_27M_SOME_FAILED	1429	43,79%

It can be seen that all 27 models have failed on 4,87% of Test data set (159 tweets). Moreover, models produced inconsistent outputs for 43,79% (1429 tweets) of the (original Kaggle) Test data set. After going trough the set of tweets, for which all models have failed, it was discovered that a lot of these tweets were possibly wrongly, or at least questionably, classified. It is obvious that in the hand classified tweets there can be mistakes. In table 3.21, there are 5 of 159 failing tweets presented. To our opinion, these tweets do not notify an emergency, but were labeled as one which do.

Table 3.21: 5 examples of failing tweets from the Test data set

Location	Keyword	Tweet	Target
MNL, Philippines	crush	omg I have a huge crush on her because of her looks. ... Not necessarily I dont think so! She is looks way be... http://t.co/xqmaTI4GI6	1
None	desolation	desolation #bored	1
None	emergency plan	Do you have an emergency drinking water plan? Download guide in English Spanish French Arabic or Vietnamese. http://t.co/S0ktilisKq	1
University of Oklahoma	sinking	"If you're lost and alone Or you're sinking like a stone. Carry on May your past be the sound Of your feet upon the ground"	1
None	hellfire	Hellfire is surrounded by desires so be careful and don let your desires control you! #Afterlife	1

Moreover, we were not able to identify any obvious feature that can be seen as "uniting" them all. They all were of different sizes, containing keywords, location, hashtags and other information. Since performing in-depth study of tweets is out of scope of this work, let us only state that if misclassified tweets are present in the Train data set, their presence may be influencing the accuracy of models in a bad way.

However, this leads us to an interesting consideration. Let us assume, for a while that inclusion of misclassified tweets is influencing the quality of results produced by the models. The one possible solution, to improve the performance of all models, could be to eliminate such tweets form the Train and Test data sets.

After looking trough the *TDS_27M* table in [22], it is easy to see that these pairs should also be analyzed separately. It is visible that some group of pairs have similar features. This groups were easily identified and therefore in google sheet [22] there were created following tables:

- *TDS_10GM_FAILED* and *TDS_10GM_SOME_FAILED* – tables with 10 pairs of models, which utilize GloVe
- *TDS_10M_FAILED* and *TDS_10M_SOME_FAILED* - tables with 10 pairs of models, which uses Keras Embeddings
- *TDS_6C_FAILED* and *TDS_6C_SOME_FAILED* - tables with 6 classifier pairs
- *TDS_4C_FAILED* and *TDS_4C_SOME_FAILED* - tables with 4 best classifier pairs (without Decision Tree and Random Forest classifier pairs, because they had the lowest scores)
- *TDS_3T_FAILED* and *TDS_3T_SOME_FAILED* - which contains top 3 best pairs (BERT, GRU and LSTM_DROPOUT)

The same, as in the case of previously discussed tables, suffix *_FAILED* in table means that it contains a set of tweets, where for every tweet, *every* pair from the group has failed to identify the correct tweet class (*yes* as *no* or *no* as *yes*). While suffix *_SOME_FAILED* means that table contains set of tweets, where for every tweet some pairs identified it correctly whereas others did not. All of these tables do not include correctly identified tweets by each pair. Summary of mentioned tables is presented in table 3.22.

Table 3.22: Summary of Grouped Predictions tables

Table Name	Tweets Count	Percentage of Test Data Set
TDS_10GM_FAILED	382	11,70%
TDS_10GM_SOME_FAILED	558	17,10%
TDS_10M_FAILED	286	8,76%
TDS_10M_SOME_FAILED	960	29,42%
TDS_6C_FAILED	347	10,63%
TDS_6C_SOME_FAILED	796	24,39%
TDS_4C_FAILED	454	13,91%
TDS_4C_SOME_FAILED	422	12,93%
TDS_3T_FAILED	334	10,23%
TDS_3T_SOME_FAILED	492	15,07%

In table 3.22, it can be seen that the combination of top three pairs had problems to identify 25,30% of the test data set (826 tweets), while they completely failed on 10,23% (334 tweets), and had different predictions on 15,07% (492) of tweets. This is the lowest sum of failed and some failed tweets counts from table 3.22. It is interesting that, in average, each pairs' group has failed completely to identify correct classes for 11% of test data set (around 360 tweets), but all 27 pairs failed on 4,87% (159) tweets.

Exploring the results further, after going through the Test data set, it was discovered that there are some tweets for which only BERT pair has failed to predict correctly while all other pairs did not fail and vice versa. This means that each pair's result can possibly improve the BERT pair's result. Therefore each of pairs can be useful.

3.8. Combined Networks, Simple Formulas Approach

In section 3.6 we have trained multiple NNs on different pairs' (model and preprocessing algorithm) predictions combinations. Unfortunately, results, from such approach, were not good enough to beat a single BERT score achieved in table 3.17. In section 3.6 it was concluded that trained NNs were not able to understand that BERT predictions should influence more than other models. Thus, knowing that BERT predictions were more accurate than other models, it was tried to achieve better results by applying different weights formulas to prediction values from different pairs. In table 3.23 there are presented formulas, which produced top results.

Table 3.23: Formulas which produced better results

ID	Pairs Group	Formula	T Win	TDS %	B Impact
E1	2GM + B	(4*B+2*GM1+1*GM2)/7	6	0,1838%	57,14%
E2	5GM + B	(5*B+(5GM))/10	8	0,2451%	50%
E3	10GM + B	(13*B+(10GM))/23	3	0,0919%	56,52%
E4	10GM + 10M + B	(22*B+(10GM)+(10M))/42	16	0,4903%	52,38%
E5	10GM + 10M + B	(33*B+2*(10GM)+(10M))/63	13	0,3984%	52,38%
E6	10GM + 10M + 6C + B	(31*B+(10GM)+(10M)+(6C))/57	17	0,5209%	54,38%
E7	E6 + E4 + E5	(2*E6+2*E4+1*E5)/5	20	0,6129%	53,18%

Shortcuts, used in table 3.23, have following meanings:

- GM – model with GloVe Embeddings
- M – model with Keras Embeddings

- C – classifier
- 10GM – predictions of 10 pairs of models with GloVe Embeddings
- 10M – predictions of 10 pairs of models with Keras Embeddings
- 6C – predictions of 6 classifier pairs
- B – prediction of BERT pair
- T Win – Tweets win
- TDS % – percentage gain on the Test data set
- B Impact – BERT pair impact in given formula

Table 3.23 demonstrates that using simple formulas, which calculates the final prediction value by applying weights to pairs predictions, resulted in being able to recognized 20 tweets more than by standalone BERT pair (from 2728 to 2748 tweets out of 3263). This is 0,6129% of the test data set. This result has beaten an additional simple NN approach, which was described in section 3.6. In all formulas, in table 3.23, BERT pair impacts the final prediction value by 50% - 58%. These formulas were established experementaly, by increasing BERT model prediction impact step by step.

This result confirms our hypothesis that NN was not able to recognize that BERT prediction is more valuable than others.

Despite the fact that all 27 pairs has predicted differently for 43,79% (section 3.7, table 3.20) of the Test data set, it seems that applying weights formulas (3.23) to 27 pairs predictions can help to create a best final results. At the same time top 3 pairs has predicted differently for only 15,07% (3.22) of the Test data set, but their final result was not as successful as of 27 pairs. Obviously, this result is specific for the Kaggle dataset under consideration, but it seems to have some bearing for the other similar tweet-related investigations. This results confirms findings in 3.7 that each of 27 pairs' results might be useful.

It can be said, paraphrasing the original paper [28], that 1 head is good, 3 is better and 27 is the best. But the effort of creating lots of models should be considered before doing so, as the gain was less than 0,65% of the Test data set. In some cases this might be crucial. In this particular competition our gain of 0,6129% plus BERT pair accuracy which was used for above's calculations 83,6040%, would be able to position our work at **46th** place at the Kaggle Leaderboard in comparison to 79th place which was achieved by standalone BERT model in the table 3.17.

3.9. Prediction Analysis of Train Data Set

After discovering, in section 3.7, table 3.20, that there are 4,87% of Test Data Set that none of 27 pairs (of model/classifier and preprocessing algorithms from table 3.18) were able to recognize correctly, it was decided to find the specific tweets in the Train data set. For this purpose each of 27 pairs were trained on the Train data set using 10-fold cross validation approach. Based on predictions for validation sets (different 10% of Train data set at each fold), a list of predictions was built for the whole Train data set. Scores of 27 pairs on the Train data set can be examined in table 3.24. Detailed results can be found in the google sheet documents [20] and [21]. Along with 10-fold cross validation, for the Train Data Set, there were also calculated predictions for the Test Data Set at each fold. So each pair has predicted each tweet from the Test Data Set 10 times. That gives us in total 3263 * 10 predictions, where 3236 is size of Test Data Set (1.2). It was discovered that, on some folds, some tweets from the Test Data Set were not recognized correctly while on others they were. For example, there were situations when the pair recognized a tweet 5 times correctly, and 5 times it failed to do it. The ratio was different depending on a tweet. The average accuracy scores were calculated for all 27 pairs on the Test data set, and are presented in table 3.24 as well.

Table 3.24: 27 pairs scores with 10 fold cross validation

Model	Algorithm ID	Train (7613)	Test (3263)
BERT	2e359f0b	0.84054	0.82908
LSTM-GloVe	8b7db91c	0.81492	0.80637
LSTM_DROPOUT-GloVe	8b7db91c	0.81978	0.81122
BI_LSTM-GloVe	8b7db91c	0.81610	0.80855
LSTM_CNN-GloVe	8b7db91c	0.81059	0.80052
FASTTEXT-GloVe	8b7db91c	0.81821	0.80818
RCNN-GloVe	71bd09db	0.81532	0.80561
CNN-GloVe	d3cc3c6e	0.81138	0.80208
RNN-GloVe	7bc816a1	0.81571	0.80509
GRU-GloVe	8b7db91c	0.82320	0.81312
HAN-GloVe	a85c8435	0.82070	0.80708
LSTM	71bd09db	0.80349	0.79372
LSTM_DROPOUT	d3cc3c6e	0.80454	0.79669
BI_LSTM	8b7db91c	0.80494	0.79255

Continuation of Table 3.24			
Model	Algorithm ID	Train (7613)	Test (3263)
LSTM_CNN	8b7db91c	0.79929	0.78694
FASTTEXT	b054e509	0.81610	0.80242
RCNN	7bc816a1	0.80284	0.78284
CNN	b054e509	0.80533	0.79764
RNN	1258a9d2	0.80888	0.79160
GRU	b054e509	0.80139	0.78765
HAN	a85c8435	0.80599	0.79678
RIDGE	2e359f0b	0.80796	0.80444
SVC	4c2e484d	0.81689	0.80251
LOGISTIC_REGRESSION	8b7db91c	0.81229	0.80245
SGD	2e359f0b	0.80704	0.80165
DECISION_TREE	7bc816a1	0.77144	0.76534
RANDOM_FOREST	60314ef9	0.79771	0.78835

The summary of distribution of tweets, for which each pair has failed to predict correctly, and for which only some pairs have failed is presented in table 3.25.

Table 3.25: Summary of 27 pairs tables were Train data set was 7613 tweets

Table Name	Tweets Count	Percentage
7000_10FCV_27M	7613	100%
7000_10FCV_27M_FAILED	347	4,56%
7000_10FCV_27M_SOME_FAILED	3281	43,10%
7000_10FCV_27M_TDS	32630	100%
7000_10FCV_27M_TDS_FAILED	1616	4,95%
7000_10FCV_27M_TDS_SOME_FAILED	14435	44,24%

Table naming can be summarized as follows:

- *7000_10FCV_27M* – 7000 means approximate Train data set size, 10FCV means 10-fold cross validations was used, 27M means that predictions for 27 pairs are presented in the table.
- *_TDS* – means that the table contains predictions for the Test Data Set.

- *_FAILED* – means that table contains only tweets for which all 27 pairs have failed to recognize them.
- *_SOME_FAILED* – means that table contains only tweets for which some pairs have failed to recognize them. This table does not contain tweets which all pairs have recognized correctly.

The first group of tables, with the prefix *7000_10FCV_27M*, represents the Train data set, while the group prefixed with *7000_10FCV_27M_TDS*, contains tweets predictions for the Test data set at each fold (3263 tweets * 10). There are 4,56% of tweets from the Train data set, which *none* pair has guessed, whereas there are 43,10% of tweets, which some pairs has guessed correctly. Nearly the same distribution was discovered with the Test data set. It was found that 1616 tweets out of 32630 tweets from the Test data set were not recognized by any pair, which gives us 161.6/3263 = 4,95% of the Test data set. This number confirms the number received in table 3.20. The numbers from 3.25 and 3.20, in general, are very close. It can be concluded that, in the Train and the Test data sets, there are around 4-5% of not recognizable tweets by any of 27 models and around 43-44% of tweets were some models fail to recognize. All other tweets are recognized by each model.

After the set of tweets, from the Train data set, for which all 27 pairs have failed to recognize, was found, it was decided to explore their influence. Specifically, we have removed this set of tweets, from the Train data set, and retrained the 27 pairs using the same approach, without these "peculiar tweets". It was done to answer the question: would it help if these tweets were not present in the Train data set in the first place? Our assumption was that these tweets might have a "bad influence" on the training process.

After removing 347 failing tweets (table 3.25) from the Train data det, the size of Train data set was reduced from 7613 to 7266 tweets. Using the same approach, as in the previous experiment, the Train data set predictions were built based on 27 pairs predictions from validation sets (different 10% of Train data set at each fold). But, in order to compare obtained pairs' accuracy to the previous experiment table 3.24, it was needed at each fold to check how these pairs performed on the 347 failing tweets. In table 3.26 there is summary of pairs' accuracy calculated by extracting failing tweets from the Train data set.

Table 3.26: 27 pairs scores where failed tweets were extracted from the Train data det

Model	Algorithm	Train(7266)	Failed(347)	Train+Failed	Test(3263)
BERT	2e359f0b	0.83646	0.02161	0.83745	0.83601

Continuation of Table 3.26					
Model	Algorithm	Train(7266)	Failed(347)	Train+Failed	Test(3263)
LSTM-GloVe	8b7db91c	0.81913	0.00548	0.81937	0.80797
LSTM_DROPOUT-GloVe	8b7db91c	0.82175	0.00432	0.82195	0.81082
BI_LSTM-GloVe	8b7db91c	0.81873	0.00346	0.81889	0.80754
LSTM_CNN-GloVe	8b7db91c	0.81348	0.00605	0.81375	0.80405
FASTTEXT-GloVe	8b7db91c	0.82320	0.00144	0.82326	0.80515
RCNN-GloVe	71bd09db	0.81821	0.00548	0.81846	0.80411
CNN-GloVe	d3cc3c6e	0.82004	0.00692	0.82036	0.80653
RNN-GloVe	7bc816a1	0.81663	0.00663	0.81693	0.80613
GRU-GloVe	8b7db91c	0.82385	0.00202	0.82395	0.80916
HAN-GloVe	a85c8435	0.82464	0.00403	0.82483	0.80803
LSTM	71bd09db	0.81229	0.00461	0.81250	0.79789
LSTM_DROPOUT	d3cc3c6e	0.81335	0.00173	0.81342	0.79721
BI_LSTM	8b7db91c	0.80599	0.00951	0.80642	0.79497
LSTM_CNN	8b7db91c	0.81059	0.00403	0.81077	0.79378
FASTTEXT	b054e509	0.81952	0.00115	0.81957	0.80328
RCNN	7bc816a1	0.81006	0.00288	0.81019	0.79074
CNN	b054e509	0.81729	0.00029	0.81730	0.80429
RNN	1258a9d2	0.81505	0.00317	0.81520	0.79690
GRU	b054e509	0.80954	0.00605	0.80981	0.79341
HAN	a85c8435	0.81518	0.00432	0.81538	0.79736
RIDGE	2e359f0b	0.81610	0.00173	0.81618	0.80141
SVC	4c2e484d	0.81624	0.00259	0.81635	0.80135
LOGISTIC_REGRESSION	8b7db91c	0.81781	0.00346	0.81797	0.80172
SGD	2e359f0b	0.81308	0.00692	0.81340	0.80015
DECISION_TREE	7bc816a1	0.78484	0.03285	0.78634	0.77346
RANDOM_FOREST	60314ef9	0.80389	0.00490	0.80411	0.79004

Comparing tables 3.24 and 3.26, it can be seen that for some pairs the average score of Test data set has increased, while for some it has decreased. The highest gain for the Test data set were obtained by the BERT pair – +0,00693, RCNN – +0,00790 and DECISION_TREE – +0,00812. For 9 pairs, the average Test data set scores were reduced, while for 18 others, the scores were enlarged. This data is summarized in table 3.27. At the same time the scores for the training data set was decreased only for 2 pairs: the BERT model pair and for the SVC classifier pair.

Table 3.27: 27 Pairs Scores Difference on the Test Data Set

Model	Algorithm ID	Train Data Score Diff	Test Data Score Diff
BERT	2e359f0b	-0.00309	+0.00693
LSTM-GloVe	8b7db91c	+0.00445	+0.00160
LSTM_DROPOUT-GloVe	8b7db91c	+0.00217	-0.00040
BI_LSTM-GloVe	8b7db91c	+0.00279	-0.00101
LSTM_CNN-GloVe	8b7db91c	+0.00316	+0.00353
FASTTEXT-GloVe	8b7db91c	+0.00505	-0.00303
RCNN-GloVe	71bd09db	+0.00314	-0.00150
CNN-GloVe	d3cc3c6e	+0.00898	+0.00445
RNN-GloVe	7bc816a1	+0.00122	+0.00104
GRU-GloVe	8b7db91c	+0.00075	-0.00396
HAN-GloVe	a85c8435	+0.00413	+0.00095
LSTM	71bd09db	+0.00901	+0.00417
LSTM_DROPOUT	d3cc3c6e	+0.00888	+0.00052
BI_LSTM	8b7db91c	+0.00148	+0.00242
LSTM_CNN	8b7db91c	+0.01148	+0.00684
FASTTEXT	b054e509	+0.00347	+0.00086
RCNN	7bc816a1	+0.00735	+0.00790
CNN	b054e509	+0.01197	+0.00665
RNN	1258a9d2	+0.00632	+0.00530
GRU	b054e509	+0.00842	+0.00576
HAN	a85c8435	+0.00939	+0.00058
RIDGE	2e359f0b	+0.00822	-0.00303
SVC	4c2e484d	-0.00054	-0.00116
LOGISTIC_REGRESSION	8b7db91c	+0.00568	-0.00073
SGD	2e359f0b	+0.00636	-0.00150
DECISION_TREE	7bc816a1	+0.01490	+0.00812
RANDOM_FOREST	60314ef9	+0.00640	+0.00169

It can be concluded that locating the failing tweets and removing them from the Train data set may be helpful to gain better results.

After training 27 pairs on the Train data set reduced by failing tweets (7266 tweets) it was expected that there would be no more tweets were all 27 pairs have failed to recognize again. This is because, the “bad tweets” were already eliminated (347 tweets). But it was discovered that there materialized a new set of tweets for which all 27 pairs have failed to classify. The summary of a new distribution for the Train and Test data sets are presented in table 3.28.

Table 3.28: Summary of 27 pairs tables were Train Data Set was 7266 tweets

Table Name	Tweets Count	Percentage
7000_10FCV_27M	7266	100%
7000_10FCV_27M_FAILED	91	1,25%
7000_10FCV_27M_SOME_FAILED	2912	40,08%
7000_10FCV_27M_TDS	32630	100%
7000_10FCV_27M_TDS_FAILED	1859	5,70%
7000_10FCV_27M_TDS_SOME_FAILED	12925	39,61%

Comparing 3.28 to 3.25 it was found that for the Train data set there occurred new 91 tweets, for which all 27 pairs have failed to classify, but the number were some pairs have failed decreased from 3281 to 2912. It can be concluded that pairs, after removing failing tweets, became more sure about 369 tweets. If considering the number of failed tweets from the Test data set, it has increased from 1616 to 1859 out of 32630 (0,75%). But the same as for the Train data set, the number of tweets were some pairs have failed decreased from 14435 to 12925 (4,63%). So as well it can be concluded that these pairs were more sure about some tweets.

Taking into account conclusions from tables 3.27, 3.25 and 3.28 it can be said that this method has a positive effect and makes models more sure in their decisions.

In the one of the last experiments, it was tried to train a model pair only on these 347 failing tweets and see how this model will perform. For that purpose the BERT pair (from table 3.18) was selected, as it proved to be the most effective one. Using 10-fold cross validation approach, it was discovered that the pair was trained successfully and was able to recognize, in average, 91% of failing tweets from the validation set at each fold. But this pair performed very poorly for tweets from the Train data set and Test data set. This results are summarized in table 3.29. From the results we have concluded that is is possible that at least some of these failing tweets are poorly or questionable classified, as was discussed already in section 3.7. Due to the fact that this is a hand classified data set and it can contain mistakes. So it could mean that BERT pair predict some of them correctly but they have wrong correct labels.

Table 3.29: Averages scores of BERT pair trained on 347 failing tweets

Set	Score
Validation Set (10% of 347)	0,91084
Train Data Set (7266)	0.38947
Test Data Set (3263)	0.41593

Note that these results immediately open an avenue for future research (especially for very large data sets), trying to "cluster" tweets and establish what makes the "bad tweets" bad in the context of given data set. This suggestion is further supported in the next section. However, such work was out of scope of this Thesis.

3.10. Prediction Analysis of Train + Test Data Set

Taking into account that the whole data set in Kaggle is, actually, tagged, we have decided to skip their competition-oriented setup and treat it as a whole. Hence, the same approach as in section 3.9 was tried on the whole data set of 10876 tweets (Train + Test data set). For that purpose 27 pairs (of model/classifier and preprocessing algorithms from table 3.18) were trained on this combined data set. With help of 10-fold cross validation 27 pairs' predictions list were build based on validation sets from each fold (different 10% of combined data set at each fold). Detailed results can be found in the google sheet documents [20] and [21] as these prediction tables have 10876 rows they cannot be included in this Thesis. In table 3.30, results for the 27 pairs on the Train + Test Data Set from [20] and [21].

Table 3.30: 27 Pairs Scores on the Train + Test Data Set

Model	Algorithm ID	Train + Test (10876)
BERT	2e359f0b	0.83781
LSTM-GloVe	8b7db91c	0.82190
LSTM_DROPOUT-GloVe	8b7db91c	0.82576
BI_LSTM-GloVe	8b7db91c	0.82365
LSTM_CNN-GloVe	8b7db91c	0.81822
FASTTEXT-GloVe	8b7db91c	0.82300
RCNN-GloVe	71bd09db	0.81712
CNN-GloVe	d3cc3c6e	0.81841
RNN-GloVe	7bc816a1	0.82034

Continuation of Table 3.30		
Model	Algorithm ID	Train + Test (10876)
GRU-GloVe	8b7db91c	0.82944
HAN-GloVe	a85c8435	0.82659
LSTM	71bd09db	0.80774
LSTM_DROPOUT	d3cc3c6e	0.81124
BI_LSTM	8b7db91c	0.80710
LSTM_CNN	8b7db91c	0.80728
FASTTEXT	b054e509	0.81776
RCNN	7bc816a1	0.80388
CNN	b054e509	0.81142
RNN	1258a9d2	0.80967
GRU	b054e509	0.80783
HAN	a85c8435	0.80976
RIDGE	2e359f0b	0.81271
SVC	4c2e484d	0.81583
LOGISTIC_REGRESSION	8b7db91c	0.81243
SGD	2e359f0b	0.81252
DECISION_TREE	7bc816a1	0.77087
RANDOM_FOREST	60314ef9	0.79947

Similarly to 3.25, the distribution of tweets, for which all pairs have failed to classify, and for tweets where some pairs have failed to classify, was obtained and is presented in table 3.31.

Table 3.31: Summary of 27 models predictions for Train + Test Data Set

Table Name	Tweets Count	Percentage of Test Data Set
10000_10FCV_27M	10876	100%
10000_10FCV_27M_FAILED	539	4,96%
10000_10FCV_27M_SOME_FAILED	3281	30,16%

There are around 4.96% (539) of not recognizable tweets by any of 27 pairs and 30,16% (3281) of tweets were some pairs fails to recognize. Other tweets were correctly classified by each pair.

The 27 pairs were retrained with the same approach as previously, on the data set consisting of Train and Test data sets, excluding the 539 failing tweets. The results of this experiment are presented in table 3.32.

Table 3.32: 27 Pairs Scores on the Train + Test Data Set without Failing Tweets

Model	Algorithm ID	Train (10337)	Failed (539)	Train + Failed
BERT	2e359f0b	0.88923	0.01911	0.84611
LSTM-GloVe	8b7db91c	0.86602	0.00371	0.82328
LSTM_DROPOUT-GloVe	8b7db91c	0.87269	0.00167	0.82952
BI_LSTM-GloVe	8b7db91c	0.86602	0.00557	0.82337
LSTM_CNN-GloVe	8b7db91c	0.86157	0.00779	0.81925
FASTTEXT-GloVe	8b7db91c	0.86853	0.00686	0.82583
RCNN-GloVe	71bd09db	0.86215	0.00538	0.81969
CNN-GloVe	d3cc3c6e	0.86485	0.00724	0.82235
RNN-GloVe	7bc816a1	0.86505	0.00371	0.82236
GRU-GloVe	8b7db91c	0.87114	0.00260	0.82810
HAN-GloVe	a85c8435	0.87172	0.00538	0.82879
LSTM	71bd09db	0.85847	0.01169	0.81650
LSTM_DROPOUT	d3cc3c6e	0.85673	0.00798	0.81467
BI_LSTM	8b7db91c	0.85518	0.00946	0.81327
LSTM_CNN	8b7db91c	0.85692	0.00853	0.81488
FASTTEXT	b054e509	0.86708	0.00074	0.82414
RCNN	7bc816a1	0.85412	0.00946	0.81226
CNN	b054e509	0.86311	0.00761	0.82072
RNN	1258a9d2	0.85421	0.00761	0.81226
GRU	b054e509	0.85653	0.00798	0.81448
HAN	a85c8435	0.85518	0.00798	0.81319
RIDGE	2e359f0b	0.86186	0.00223	0.81925
SVC	4c2e484d	0.86234	0.00260	0.81973
LOGISTIC_REGRESSION	8b7db91c	0.86350	0.00130	0.82077
SGD	2e359f0b	0.85895	0.00353	0.81656
DECISION_TREE	7bc816a1	0.83090	0.01670	0.79055
RANDOM_FOREST	60314ef9	0.84638	0.00408	0.80463

Comparing tables 3.30 and 3.32, it can be seen that for the BI_LSTM-GloVe pair and the GRU-GloVe pair, the average Train data score have decreased. For the remaining 25 pairs, this method of removing failing tweets was beneficial. This data is presented in table 3.33. Quite similar results were achieved in table 3.27. But in the 3.27 the other two models had decreased score.

Table 3.33: 27 Models Scores Difference

Model	Algorithm ID	Train Data Score Diff
BERT	2e359f0b	+0.00830
LSTM-GloVe	8b7db91c	+0.00138
LSTM_DROPOUT-GloVe	8b7db91c	+0.00376
BI_LSTM-GloVe	8b7db91c	-0.00028
LSTM_CNN-GloVe	8b7db91c	+0.00103
FASTTEXT-GloVe	8b7db91c	+0.00283
RCNN-GloVe	71bd09db	+0.00257
CNN-GloVe	d3cc3c6e	+0.00394
RNN-GloVe	7bc816a1	+0.00202
GRU-GloVe	8b7db91c	-0.00134
HAN-GloVe	a85c8435	+0.00220
LSTM	71bd09db	+0.00876
LSTM_DROPOUT	d3cc3c6e	+0.00343
BI_LSTM	8b7db91c	+0.00617
LSTM_CNN	8b7db91c	+0.00760
FASTTEXT	b054e509	+0.00638
RCNN	7bc816a1	+0.00838
CNN	b054e509	+0.00930
RNN	1258a9d2	+0.00259
GRU	b054e509	+0.00665
HAN	a85c8435	+0.00343
RIDGE	2e359f0b	+0.00654
SVC	4c2e484d	+0.00390
LOGISTIC_REGRESSION	8b7db91c	+0.00834
SGD	2e359f0b	+0.00404
DECISION_TREE	7bc816a1	+0.01968
RANDOM_FOREST	60314ef9	+0.00516

In summary, it can be stated that locating “failing tweets”, and removing them from the data set may bring some advantages. This might be explained by the assumption that these tweets, which always fail are influencing in a bad way model calibration during the training process. Finally, it may be also possible to explore a possibility of using models to “cluster tweets”, then establishing the best approach to each cluster. Next, for the “new”/“unknown” tweets, one could try to establish which cluster they are the closest to (using, for instance, some NLP-based approaches) and use best model for this cluster to establish, which category the incoming tweet belong to. However, this is clearly out of scope of this work. Nevertheless, establishing potential of this direction of work is one of contributions of this Thesis.

3.11. Concluding remarks

Let us now summarize the main conclusions for the research concluded within the scope of this Thesis.

- In section 3.1 ten data preprocessing algorithms were selected. All these algorithms had same 7 out of 17 frozen configuration parameters, which were selected experimentally. Changing some of the frozen parameters decreased the accuracy of the tested models severely. These algorithms were selected in order to test the remaining 10 of 17 non frozen parameters.

  Five of these algorithms (1258a9d2, 60314ef9, 4c2e484d, 8b7db91c and 7bc816a1) were configured to test the the "flag presence parameters" group (table 3.1). Other parameters for these preprocessing algorithms have the same values (tables 3.2 and 3.3).

  Four of these algorithms (a85c8435, b054e509, d3cc3c6e and 1258a9d2) have similar configuration for the "flag presence parameters" group (table 3.1). Four parameters from this group are set to True, because these flags have a vector representation in GloVe. These algorithms have different configurations for other parameters groups (tables 3.2 and 3.3).

  Three of these algorithms (2e359f0b, 71bd09db and 7bc816a1) have the same configuration values (all set to False) for the "flag presence parameters" group in table 3.1 and different configurations for other parameters groups (tables 3.2 and 3.3).

  In that way these ten algorithms allowed us to test the ten non frozen parameters.

- In section 3.2 six classifiers with different vectorizers and data preprocessing algorithms were tested. It was discovered that the best average of 10-fold cross validation experiments accuracy of 0.80444 among classifiers was achieved by the Ridge classifier, utilizing Tfidf vectorizer with [1, 2] ngrammar range and with two preprocessing lgorithms: 2e359f0b and 71bd09db (table 3.4). These two algorithms are very similar except 2e359f0b does not remove punctuation from tweets while 71bd09db does. The top three classifiers were Ridge, SVC and Logistic Regression. While the worst results were achived with the Decission Tree and Random Forest classifiers. Top three preprocessing algorithms were 71bd09db, 8b7db91c and 60314ef9 which received the same relative score (table 3.11), but it cannot be said that one of them is the best one and every classifier performs the best with it.

- In section 3.3 ten NN models based on Keras embedddings were tested. It was found out that FastText NN model had the best average score of 0.803739 calculated from 10-fold cross validation experiments with the b054e509 data preprocessing algorithm (table 3.12). Overall, the highest result of 0.809378 was achieved by the CNN NN model with 2e359f0b preprocessing algorithm on the 7th fold counting from 0 (table 3.13). It can be said that all models average scores were close to each other and were achieved by different preprocessing algorithm. Thus, the same as for classifiers, there was no best data preprocessing algorithm found which was performing best with all NN models.
- In section 3.4 ten NN models based on GloVe embeddings were tested. All NN models with these embeddings produced better scores than same models with Keras embeddings in section 3.3. The top average score 0.814435 of 10-fold cross validation experiments were achieved by GRU NN model in combination with 8b7db91c data preprocessing algorithm (table 3.14). This NN model also had the top highest score achieved (0.821943) but with the a85c8435 preprocessing algorithm (table 3.15). Similarly to 3.3, all NN models obtained quite similar results, but in this case it could be said that the 8b7db91c data preprocessing algorithm was standing out among others. It produced 5 top averages results and 4 top highest results out of 10 NN models.
- In section 3.5 BERT model results were described. BERT model had the best average score among other NN models and classifiers. The best average score 0.834692 was achieved with the 2e359f0b data preprocessing algorithm (table 3.16). The best top score was achieved with the same preprocessing algorithm, and was 0.840638 (table 3.17). This score positioned our work at 79 place out of 3402 submissions in the Kaggle Leaderboard if do not include the scores which are better than 98%. Here, we have assumed that these submissions used leaked data in their training process.
- In section 3.6 the 27 pairs' (of classifiers and NN modes and their best preprocessing algorithm from table 3.18) predictions combination approach was described. These 27 pairs were used to produce predictions input to train a simple NN network in order to receive a better score. This approach did overcome single BERT model accuracy by 0.4%.

Better gain was achieved when models excluding BERT with similar accuracy were used. With the combination of 10 NN models which uses GloVe embeddings and 6 classifiers from table 3.18 the 1.5%-gain was achieved with some NNs on some folds. It was concluded that these NN model were not able to find out that BERT predictions should have a bigger impact than other models for final predictions.

- In section 3.7 the Test data set was analysed based on 27 pairs (from table 3.18). A set of tweets were all models have failed to classify correctly was discovered. Along with mentioned discovery, it was found that there were tweets were only BERT pair has failed to predict correctly while all other models did not and vice versa. It was concluded that each pair's result can possibly improve the BERT pair's result. Therefore each of pairs can be useful.

- In section 3.8, by applying simple formulas to calculate a final predictions result, it was gained 20 tweets more than with stand alone BERT pair. These formulas were more effective than NN approach from section 3.6. The best formula which produced the best gain was achieved when all 27 pairs were involved, what confirms our conclusion from section 3.7. This results would be able to position our work at **46th** place at the Kaggle Leaderboard in comparison to 79th place which was achieved by the standalone BERT model in table 3.17 from section 3.5.

- In section 3.9 Train data set was analysed. In this set tweets, which failed for *all* 27 pairs were found. Later these tweets were removed from the data set and the 27 pairs were trained on this data. It was discovered that this approach produced better results for a significant amount of models.

- In section 3.10 the combination of Train and Test data was analysed. The same as in 3.9 after discovering failing tweets for 27 pairs, these tweets were removed from the data set and 27 pairs were trained on the new data. This experiment as well as in 3.9 confirmed that the approach of removing tweets which have failed with all pairs was beneficial almost for all pairs.

Finally, the main result of this Thesis is that treating tweets on the syntactic level alone, which is what is actually done by all of the tried methods is not enough. Taking into account obtained results, it can be stipulated that NLP-based approaches (and, possibly, semantic technologies) may bring about better results in tweet classification.

Bibliography

[1] O. Ajao, D. Bhowmik, and S. Zargari. Fake news identification on twitter with hybrid cnn and rnn models. in proceedings of the international conference on social media & society. *SMSociety*, 2018. DOI: 10.1145/3217804.3217917.

[2] Z. Ashktorab, C. Brown, M. Nandi, and C. Mellon. Tweedr: Mining twitter to inform disaster response. Proceedings of the 11th International ISCRAM Conference University Park, May 2014.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, May 2019. arXiv: 1810.04805.

[4] S. ES. Basic eda, cleaning and glove. `https://www.kaggle.com/shahules/basic-eda-cleaning-and-glove`.

[5] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37, Mar 1996. DOI: 10.1609/aimag.v17i3.1230.

[6] M. Ganzha, M. Paprzycki, and J. Stadnik. Combining information from multiple search engines - preliminary comparison. *Information Sciences*, 180(10):1908–1923, 2010. DOI: 10.1016/j.ins.2010.01.010.

[7] Google. Bert github. `https://github.com/google-research/bert`.

[8] Google. Tensorflow hub. `https://www.tensorflow.org/hub`.

[9] Kaggle. Kaggle competition leaderboard. `https://www.kaggle.com/c/nlp-getting-started/leaderboard`.

[10] Kaggle. Kaggle competition: Real or not? nlp with disaster tweets. predict which tweets are about real disasters and which ones are not. `https://www.kaggle.com/c/nlp-getting-started/overview`.

[11] Keras. Keras embedding layer. `https://keras.io/api/layers/core_layers/embedding/`.

[12] Keras. Keras library. https://keras.io/.

[13] Keras. Sequential model. https://keras.io/api/models/sequential/.

[14] U. Kursuncu, M. Gaur, K. T. Usha Lokala, A. Sheth, and I. Budak Arpinar. Predictive analysis on twitter: Techniques and applications. Kno.e.sis Center, Wright State University, Jun 2018. arXiv: 1806.02377.

[15] G. Ma. Tweets classification with bert in the field of disaster management. Department of Civil Engineering, Stanford University, 2019.

[16] NLTK. Nltk library. https://www.nltk.org/.

[17] NLTK. Nltk porterstemmer. https://www.nltk.org/api/nltk.stem.html#module-nltk.stem.porter.

[18] NLTK. Nltk tweettokenizer. https://www.nltk.org/api/nltk.tokenize.html#module-nltk.tokenize.casual.

[19] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. https://nlp.stanford.edu/projects/glove/.

[20] M. Plakhtiy. 27 pairs' results for the train and train + test data sets. https://docs.google.com/spreadsheets/d/1XlrYNnOHGkKTR7tTocK1dpIkzOYoQcuuSg-whnbF8to/edit?usp=sharing.

[21] M. Plakhtiy. 27 pairs' results for the train and train + test data sets, without failing tweets. https://docs.google.com/spreadsheets/d/1yCrIO7c6KpHO7rzkJ2rr5wkQvTJHElGzg0dhiNfhPZs/edit?usp=sharing.

[22] M. Plakhtiy. Results on google drive. https://docs.google.com/spreadsheets/d/1eP0DdEMxzNLT6ecfdN5Kf5ctK1BSYNgoq1YHylSVDLc/edit?usp=sharing.

[23] R. Rohith. In-depth guide to google's bert. https://www.kaggle.com/ratan123/in-depth-guide-to-google-s-bert.

[24] SkLearn. Count vectorizer. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html.

[25] SkLearn. Scikit-learn library user guide. https://scikit-learn.org/stable/user_guide.html.

[26] SkLearn. Stratifiedkfold. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html`.

[27] SkLearn. Tfidf vectorizer. `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html`.

[28] J. Stadnik, M. Ganzha, and M. Paprzycki. Are many heads better than one – on combining information from multiple internet sources. *Intelligent Distributed Computing, Systems and Applications*, pages 177–186, 2008. DOI: 10.1007/978-3-540-85257-5_18.

[29] K.-G. Thanos, A. Polydouri, A. Danelakis, D. Kyriazanos, and S. C. Thomopoulos. Combined deep learning and traditional nlp approaches for fire burst detection based on twitter posts. *IntechOpen*, April 2019. DOI: 10.5772/intechopen.85075.

[30] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy. Hierarchical attention networks for document classification. Carnegie Mellon University, Microsoft Research, Redmond, Jun 2016.

[31] C. Zhang, A. Rajendran, and M. Abdul-Mageed. Hyperpartisan news detection with attention-based bi-lstms. Natural Language Processing Lab, The University of British Columbia, 2019.

List of Figures

List of Tables

List of appendices

A. Classes Code Structures

A.1. TweetsPreprocessor Class

```python
...

class TweetsPreprocessor:
    def __init__(self, tokenizer, stemmer, stop_words, slang_abbreviations, splitters):
        self.tokenizer = tokenizer
        self.stemmer = stemmer
        self.stop_words = stop_words
        self.slang_abbreviations = slang_abbreviations
        self.slang_abbreviations_keys = slang_abbreviations.keys()
        self.splitters = splitters

    @staticmethod
    def _add_link_flag(tweet):
        ...

    @staticmethod
    def _add_user_flag(tweet):
        ...

    @staticmethod
    def _add_hash_flag(tweet):
        ...

    @staticmethod
    def _add_number_flag(tweet):
        ...
```

```
@staticmethod
def _add_keyword_flags(tweets, keywords):
    ...

@staticmethod
def _add_location_flags(tweets, locations):
    ...

@staticmethod
def _is_flag(word):
    ...

@staticmethod
def _remove_links(words):
    ...

@staticmethod
def _remove_users(words):
    ...

@staticmethod
def _remove_hash(words):
    ...

@staticmethod
def _to_lower_case(words):
    ...

@staticmethod
def _is_punctuation(word):
    ...

@staticmethod
```

```
def _is_empty_str(word):
    ...

@staticmethod
def _is_hashtag(word):
    ...

@staticmethod
def _is_number(word):
    ...
@staticmethod
def _remove_punctuations(words):
    ...

@staticmethod
def _remove_not_alpha(words):
    ...

@staticmethod
def _remove_numbers(words):
    ...

@staticmethod
def _join(words):
    ...

@staticmethod
def _split(tweet):
    ...

def _split_words(self, words):
    ...

def _tokenize(self, tweet):
    ...

def _remove_stop_words(self, words):
    ...

def _stem(self, words):
    ...

def _unslang(self, words):
    ...

def preprocess(self, tweets, options=None, keywords=None, locations=None):
    if options is None:
        options = {}

    t = tweets.map(self._tokenize)

    t = t.map(self._join)

    if options.get('add_link_flag', True):
        t = t.map(TweetsPreprocessor._add_link_flag)

    if options.get('add_user_flag', True):
        t = t.map(TweetsPreprocessor._add_user_flag)

    if options.get('add_hash_flag', True):
        t = t.map(TweetsPreprocessor._add_hash_flag)

    if options.get('add_number_flag', True):
        t = t.map(TweetsPreprocessor._add_number_flag)

    if options.get('add_keyword_flag', True) and keywords is not None:
        t = TweetsPreprocessor._add_keyword_flags(t, keywords)
    if options.get('add_location_flag', True) and locations is not None:
        t = TweetsPreprocessor._add_location_flags(t, locations)

    t = t.map(self._split)

    if options.get('remove_links', True):
        t = t.map(TweetsPreprocessor._remove_links)

    if options.get('remove_users', True):
        t = t.map(TweetsPreprocessor._remove_users)

    if options.get('unslang', True):
        t = t.map(self._unslang)

    if options.get('split_words', True):
        t = t.map(self._split_words)

    if options.get('remove_hash', True):
        t = t.map(TweetsPreprocessor._remove_hash)

    if options.get('stem', True):
        t = t.map(self._stem)

    if options.get('remove_punctuations', True):
        t = t.map(TweetsPreprocessor._remove_punctuations)

    if options.get('remove_numbers', True):
        t = t.map(TweetsPreprocessor._remove_numbers)

    if options.get('to_lower_case', True):
        t = t.map(TweetsPreprocessor._to_lower_case)

    if options.get('remove_stop_words', True):
        t = t.map(self._remove_stop_words)
    if options.get('remove_not_alpha', True):
        t = t.map(TweetsPreprocessor._remove_not_alpha)

    if options.get('join', True):
        t = t.map(TweetsPreprocessor._join)

    return t
```

A.2. Sklearn Class

```
...
class Sklearn:
    VECTORIZERS = {
        'TFIDF': TfidfVectorizer,
        'COUNT': CountVectorizer
    }

    CLASSIFIERS = {
        'RIDGE': RidgeClassifier,
        'LOGISTIC_REGRESSION': LogisticRegression,
        'RANDOM_FOREST': RandomForestClassifier,
        'DECISION_TREE': DecisionTreeClassifier,
        'SVM': SVC,
        'SGDClassifier': SGDClassifier
    }
```

A.3. Keras Class

```
...

class TestDataCallback(Callback):
    def __init__(self, x_test, y_test):
        super().__init__()
        self.accuracy = []
        self.loss = []
        self.x_test = x_test
        self.y_test = y_test

    def on_epoch_end(self, epoch, logs=None):
        score = self.model.evaluate(self.x_test, self.y_test, verbose=1)
        self.loss.append(score[0])
        self.accuracy.append(score[1])

class Keras:
    OPTIMIZERS = {
        'adam': Adam,
        'rmsprop': RMSprop
    }

    DEFAULTS = {
        'ACTIVATION': 'sigmoid',
        'OPTIMIZER': 'adam',
        'LEARNING_RATE': 1e-4,
    }

    @staticmethod
    def get_sequential_model(layers, config):
        if layers is None or config is None:
            raise ValueError('Layers and config can not be None!')

        model = Sequential()

        if config.get('EMBEDDING_OPTIONS') is not None:
            model.add(Embedding(**config['EMBEDDING_OPTIONS']))

        for layer in layers:
```

```
        model.add(layer)

    model.add(Dense(
        1,
        activation=config.get('ACTIVATION', Keras.DEFAULTS['ACTIVATION'])
    ))

    model.compile(
        optimizer=Keras.OPTIMIZERS[
            config.get('OPTIMIZER', Keras.DEFAULTS['OPTIMIZER'])
        ](
            learning_rate=config.get(
                'LEARNING_RATE',
                Keras.DEFAULTS['LEARNING_RATE']
            )
        ),
        loss='binary_crossentropy',
        metrics=['accuracy']
    )

    return model

@staticmethod
def _get_lstm_model(config):
    return Keras.get_sequential_model(
        [LSTM(config['LSTM_UNITS'])],
        config,
    )

@staticmethod
def _get_lstm_dropout_model(config):
    ...

@staticmethod
```

```
    def _get_bi_lstm_model(config):
        ...

    @staticmethod
    def _get_lstm_cnn_model(config):
        ...

    @staticmethod
    def _get_fast_text_model(config):
        ...

    @staticmethod
    def _get_rcnn_model(config):
        ...

    @staticmethod
    def _get_cnn_model(config):
        ...

    @staticmethod
    def _get_rnn_model(config):
        ...

    @staticmethod
    def _get_gru_model(config):
        ...

    @staticmethod
    def get_model(config):
        models = {
            'LSTM': Keras._get_lstm_model,
            'LSTM_DROPOUT': Keras._get_lstm_dropout_model,
            'BI_LSTM': Keras._get_bi_lstm_model,
            'LSTM_CNN': Keras._get_lstm_cnn_model,
            'FASTTEXT': Keras._get_fast_text_model,
            'RCNN': Keras._get_rcnn_model,
            'CNN': Keras._get_cnn_model,
            'RNN': Keras._get_rnn_model,
            'GRU': Keras._get_gru_model,
        }

        return models[config['TYPE']](config)

    @staticmethod
    def get_bert_model(
            bert_layer,
            input_length,
            optimizer='rmsprop',
            learning_rate=2e-6
    ):
        input_word_ids = Input(
            shape=(input_length,), dtype=tf.int32, name="input_word_ids"
        )
        input_mask = Input(shape=(input_length,), dtype=tf.int32, name="input_mask")
        segment_ids = Input(shape=(input_length,), dtype=tf.int32, name="segment_ids")

        _, sequence_output = bert_layer([input_word_ids, input_mask, segment_ids])
        clf_output = sequence_output[:, 0, :]
        out = Dense(1, activation='sigmoid')(clf_output)

        model = Model(inputs=[input_word_ids, input_mask, segment_ids], outputs=out)
        model.compile(
            optimizer=Keras.OPTIMIZERS[optimizer](learning_rate=learning_rate),
            loss='binary_crossentropy',
            metrics=['accuracy']
        )

        return model
    @staticmethod
    def _plot(history, string):
        ...

    @staticmethod
    def draw_graph(history):
        ...

    @staticmethod
    def fit(model, data, config):
        is_with_test_data = len(data) == 6

        if is_with_test_data:
            x_train, y_train, x_val, y_val, x_test, y_test = data
        else:
            x_train, y_train, x_val, y_val = data

        callbacks = []

        if is_with_test_data:
            test_data_callback = TestDataCallback(
                x_test=x_test,
                y_test=y_test
            )
            callbacks.append(test_data_callback)

        if config.get('DIR') is not None and config.get('PREFIX') is not None:
            suffix = '-e{epoch:03d}-a{accuracy:03f}-va{val_accuracy:03f}-ta.h5'
            callbacks.append(ModelCheckpoint(
                config['DIR'] + config['PREFIX'] + suffix,
                verbose=1,
                monitor='val_loss',
                save_best_only=True,
                mode='auto'
            ))

        history = model.fit(
            x=x_train, y=y_train,
            batch_size=config['BATCH_SIZE'],
            epochs=config['EPOCHS'],
            verbose=1,
            validation_data=(
                x_val,
                y_val
            ),
            callbacks=callbacks
        )

        model_history = history.history.copy()
        model_history['test_loss'] = test_data_callback.loss
        model_history['test_accuracy'] = test_data_callback.accuracy
        model_history = {
            k: [round(float(v), 6) for v in data] for k, data in model_history.items()
        }

        return model_history
```

B. Applications Code Structures

B.1. app_kfold_sklearn.py

```
...
VECTORIZERS = [
    {
        'TYPE': 'TFIDF',
        'OPTIONS': {
            'binary': True,
            'ngram_range': (1, 1)
        }
    },
    {
        'TYPE': 'COUNT',
        'OPTIONS': {
            'binary': True,
            'ngram_range': (1, 1)
        }
    },
    ...
]
CLASSIFIERS = [
    {
        'TYPE': 'RIDGE',
        'OPTIONS': {}
    },
    {
        'TYPE': 'LOGISTIC_REGRESSION',
        'OPTIONS': {}
    },
    ...
]

...

for VECTORIZER in VECTORIZERS:
    for CLASSIFIER in CLASSIFIERS:
        for algorithm_id, preprocessing_algorithm in PREPROCESSING_ALGORITHMS.items():
            ...
            kfold = StratifiedKFold(n_splits=KFOLD, shuffle=True, random_state=SEED)
            train_data['preprocessed'] = tweets_preprocessor.preprocess(
                train_data.text,
                preprocessing_algorithm,
                keywords=train_data.keyword,
                locations=train_data.location
            )
            test_data['preprocessed'] = tweets_preprocessor.preprocess(
                test_data.text,
                preprocessing_algorithm,
                keywords=test_data.keyword,
                locations=test_data.location
            )
            ...
            for train, validation in kfold.split(inputs, targets):
                vectorizer = Sklearn.VECTORIZERS[VECTORIZER['TYPE']](
                    **VECTORIZER['OPTIONS']
                )
                ...
                classifier = Sklearn.CLASSIFIERS[CLASSIFIER['TYPE']](
                    **CLASSIFIER['OPTIONS']
                )
                classifier.fit(x_train, y_train)
                ...
                log_classifier(LOG_DICT)

...
```

B.2. app__kfold__keras.py

```
...
USE_GLOVE = True

if USE_GLOVE:
    GLOVE_SIZE = 200
    GLOVE = f'glove.twitter.27B.{GLOVE_SIZE}d.txt'
    GLOVE_FILE_PATH = f'./data/glove/{GLOVE}'
    GLOVE_EMBEDDINGS = get_glove_embeddings(GLOVE_FILE_PATH)

NETWORKS_KEYS = [
    'LSTM', 'LSTM_DROPOUT', 'BI_LSTM', 'LSTCM_CNN',
    'FASTTEXT', 'RCNN', 'CNN', 'RNN', 'GRU'
]

for key in NETWORKS_KEYS:
    MODEL_CONFIG = get_model_config(key, USE_GLOVE)

    for key, preprocessing_algorithm in get_preprocessing_algorithm().items():
        CONFIG = MODEL_CONFIG.copy()
        if USE_GLOVE:
            CONFIG['GLOVE'] = {
                'SIZE': GLOVE_SIZE
            }
        CONFIG['UUID'] = str(uuid.uuid4())
        CONFIG['PREPROCESSING_ALGORITHM'] = preprocessing_algorithm
        CONFIG['PREPROCESSING_ALGORITHM_UUID'] = key
        CONFIG['KFOLD_HISTORY'] = []
```

```
kfold = StratifiedKFold(n_splits=KFOLD, shuffle=True, random_state=SEED)

train_data['preprocessed'] = tweets_preprocessor.preprocess(
    train_data.text,
    preprocessing_algorithm,
    keywords=train_data.keyword,
    locations=train_data.location
)

test_data['preprocessed'] = tweets_preprocessor.preprocess(
    test_data.text,
    preprocessing_algorithm,
    keywords=test_data.keyword,
    locations=test_data.location
)

inputs = train_data['preprocessed']
targets = train_data['target']

for train, validation in kfold.split(inputs, targets):
    keras_tokenizer = Tokenizer()

    (x_train, x_val, x_test), input_dim, input_len = Helpers.get_model_inputs(
        (inputs[train], inputs[validation], test_data.preprocessed),
        keras_tokenizer
    )
    y_train = targets[train]
    y_val = targets[validation]
    y_test = test_data.target.values

    CONFIG['EMBEDDING_OPTIONS']['input_dim'] = input_dim
    CONFIG['EMBEDDING_OPTIONS']['input_length'] = input_len

    if USE_GLOVE:
```

```
                Helpers.with_glove_embedding_options(
                    CONFIG,
                    keras_tokenizer,
                    GLOVE_EMBEDDINGS
                )

            model = Keras.get_model(CONFIG)

            history = Keras.fit(
                model,
                (x_train, y_train, x_val, y_val, x_test, y_test),
                CONFIG
            )
            ...

            CONFIG['KFOLD_HISTORY'].append(history)

            log_model(CONFIG)
```

B.3. app__kfold__bert.py

```
...
PREPROCESSING_ALGORITHMS = get_preprocessing_algorithm(join=True)
PREPROCESSING_ALGORITHMS['None'] = {}

for algorithm_id, preprocessing_algorithm in PREPROCESSING_ALGORITHMS.items():
    MODEL = {
        'UUID': str(uuid.uuid4()),
        'BERT': 'bert_en_uncased_L-24_H-1024_A-16',
        'BERT_VERSION': 1,
        'BATCH_SIZE': 16,
        'EPOCHS': 3,
        'OPTIMIZER': 'adam',
        'LEARNING_RATE': 2e-6,
        'PREPROCESSING_ALGORITHM_UUID': algorithm_id,
        'PREPROCESSING_ALGORITHM': preprocessing_algorithm,
        'KFOLD_HISTORY': []
    }

    kfold = StratifiedKFold(n_splits=KFOLD, shuffle=True, random_state=SEED)

    if algorithm_id != 'None':
        train_data['preprocessed'] = tweets_preprocessor.preprocess(
            train_data.text,
            preprocessing_algorithm,
            keywords=train_data.keyword,
            locations=train_data.location
        )

        test_data['preprocessed'] = tweets_preprocessor.preprocess(
            test_data.text,
            preprocessing_algorithm,
            keywords=test_data.keyword,
            locations=test_data.location
        )
    else:
        train_data['preprocessed'] = train_data.text
        test_data['preprocessed'] = test_data.text

    inputs = train_data['preprocessed']
    targets = train_data['target']

    k = 0

    for train, validation in kfold.split(inputs, targets):
        x_train = inputs[train]
        y_train = targets[train]
```

```
        x_val = inputs[validation]
        y_val = targets[validation]

        x_test = test_data.preprocessed
        y_test = test_data.target.values

        bert_layer = hub.KerasLayer(
            f'https://tfhub.dev/tensorflow/{MODEL["BERT"]}/{MODEL["BERT_VERSION"]}',
            trainable=True
        )
        vocab_file = bert_layer.resolved_object.vocab_file.asset_path.numpy()
        do_lower_case = bert_layer.resolved_object.do_lower_case.numpy()
        tokenizer = FullTokenizer(vocab_file, do_lower_case)



        x_train, INPUT_LENGTH = Helpers.get_bert_input(x_train, tokenizer)
        x_val = Helpers.get_bert_input(x_val, tokenizer, input_length=INPUT_LENGTH)
        x_test = Helpers.get_bert_input(x_test, tokenizer, input_length=INPUT_LENGTH)

        MODEL['INPUT_LENGTH'] = INPUT_LENGTH

        model = Keras.get_bert_model(
            bert_layer=bert_layer,
            input_length=INPUT_LENGTH,
            optimizer=MODEL['OPTIMIZER'],
            learning_rate=MODEL['LEARNING_RATE']
        )

        history = Keras.fit(
            model,
            (x_train, y_train, x_val, y_val, x_test, y_test),
            MODEL
        )
```

```
log_model(MODEL)
```