Nanyang Technological University

AI6102:
Machine Learning Methodologies & Applications
Term Paper

Kaggle Competition (Knowledge)
Real or Not? NLP with Disaster Tweets

Ron Kow Kheng Hui
Brandon Chua Shao Jie

School of Computer Science and Engineering

November 30, 2020

Abstract

Microblogging is a popular form of communication in social networking. Despite its informal form, microblogging websites such as Twitter and Weibo are also an influential medium for news reporting. Breaking news of random incidents are often disseminated on Twitter by the general public who are the first to witness these incidents. News reporters routinely monitor microblogs for any news-worthy incidents.

This project is a Kaggle challenge entitled "Real or Not? NLP with Disaster Tweets". Given a set of tweets, most of which contain words related to disasters (e.g., "earthquake", "terrorism", "emergency plan"), the objective is to build a model to determine if a tweet is about an actual disaster. Given a tweet containing words related to disasters, the model classifies the tweet into one of two classes: a disaster tweet, or not a disaster tweet.

We selected a set of word tokens from the raw data as features and experimented with different classical and ensemble classification algorithms. We achieved the best performance with a gradient tree boosting model, trained using the XGBoost library. The model achieved a F_1 score of 79.5% on the official test dataset in Kaggle.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Popular microblogging websites such as Twitter [2] and Weibo [3] provide a platform for users to communicate thoughts, ideas, and real life events in an informal and accessible manner. Microblogging is not only a popular form of social networking. It has also become an influential medium for news reporting.

The latest news are often released on official microblogs before the full detailed reports are published on official websites. These may be breaking news released by news media companies (i.e., newspapers, magazines, television cable news, etc.), or important news released by organizations (i.e., government agencies, corporations, etc.) on their official microblogging sites. For example, professional football clubs in the UK often announce signings of new players on Twitter before an official statement is released. We have also seen the trend in recent years for politicians to use Twitter to announce random policies that may interest their supporters.

Very often, the general public are the first to disseminate news about random incidents happening across the world. Within minutes of a news-worthy incident occurring, such as a road accident or a sighting of a wild animal in an urban area, for instance, people on the scene of the incident would post on Twitter or Weibo what they just witnessed. These posts are often accompanied by videos taken by mobile phones.

Thus, microblogs have become an important component of news reporting and journalism. News reporters routinely monitor microblogs for any news-worthy incidents. In most parts of the world, Twitter is the most widely used microblogging service. Every reporter or journalist today is expected to tweet regularly as part of their job, in order to build a network with their readers and also to report breaking news and news updates, or simply to retweet news from other twitter users.

Many natural and man-made disasters, such as road accidents, earthquakes and sudden explosions, occur without warning. A twitter crawler that is able to detect the latest tweets on such disasters would be a useful tool for news media companies on the hunt for breaking news.

This project is a Kaggle [1] challenge entitled "Real or Not? NLP with Disaster Tweets". A set of raw tweets is provided. Most of these tweets contain words related to disasters such as "earthquake", "terrorism", or "emergency plan". Many of these tweets are about actual disasters. But some of the tweets use words related to disasters in a different context. For example, a tweet could be about a disaster movie such as Titanic. The objective of this challenge is to build a model which is able to determine if a tweet relates to an actual disaster or not. This is a supervised learning text classification problem.

1.1 Problem Statement

Given a labeled training dataset of raw tweets containing words related to disasters, our objective is to build a model that is able to classify a tweet into one of two classes: a disaster tweet (class label 1), or not a disaster tweet (class label 0).

1.2 Organization of the Report

The rest of this report is organized into three chapters.

- In Chapter 2, we describe our solution and modelling process, and discuss the challenges of this problem.
- In Chapter 3, we describe the algorithms we use in our experiments and present the results.
- Chapter 4 concludes the report with an overall summary and the lessons we have learned.

Chapter 2

Proposed Solution and Model Development

This chapter describes the raw data and our proposed solution for developing the classification model. We also discuss the challenges this problem poses.

2.1 Proposed Solution

In machine learning, the performance of the trained model is very much affected by three factors: the data, the algorithm, and the complexity of the problem. We need to consider the size of the dataset, the quality of the data and how we process the data. This in turn affects the choice of learning algorithm or method. If deep learning methods are to be used, then the choice of the neural network architecture is crucial. Some complex problems such as image classification or machine translation work better with neural networks. For simple classification problems, a simple linear model such as Logistic Regression could perform very well.

For this project, the training dataset of 7613 data instances is relatively small. The problem of text classification is not complex. For these reasons, we will not use deep learning methods. We will only experiment with traditional machine learning algorithms, including state of the art ensemble algorithms such as gradient tree boosting.

Our proposed solution is outlined in the following steps:

1. **Raw data analysis**. This involves understanding the raw data and its inherent problems through extracting some statistics. We describe our analysis in Section 2.2.

2. **Text preprocessing**. This includes data cleaning, handling of missing values, and tokenization. We describe this process in Section 2.4
3. **Feature engineering**. We will select a *bag of words* as the set of features. This is a set of selected text tokens which we believe are important features. The bag of words representation is explained in Section 2.3. We will justify our selection of features in Section 2.5.
4. **Feature values**. After selecting the features, we will prepare the training and test datasets of feature values. As each data instance is a short text string, we will use one-hot encoding for the feature values. This is explained in Section 2.6
5. **Experiments with algorithms**. We will train different models using different basic and ensemble algorithms. As the training dataset is quite small, we will not split it to produce a validation set. Each trained model will be evaluated using five-fold cross-validation. The best performing model will be our baseline model. We describe our experiments and present our results in Sections 3.1 and 3.2.
6. **Experiments with XGBoost**. We will experiment with gradient boosted trees from the XGBoost library [5] [4]. We will tune the algorithm to obtain the best possible model. As before, we will use five-fold cross-validation to evaluate our models. We present our results in Section 3.3.
7. **Submission of test data predictions to Kaggle**. Finally, we will make predictions on the test dataset using our best model and submit our predictions to Kaggle to see the test results. We present our test score on the Kaggle leaderboard in Section 3.3.3.

2.2 Raw Data Analysis

The training dataset from Kaggle consists of 7613 raw tweets, most of which contain words related to disasters. There are five columns in the dataset, for these five fields: *id*, *keyword*, *location*, *text*, and *target*.

The entire string of a raw tweet is stored in the *text* field. Twitter currently limits the length of each tweet to 140 characters. Many of the raw tweets contain URLs, hashtags prefixed by the # symbol (e.g., `#CNN`, `#interracial`, `#GlobalWarming`), and usernames prefixed by the @ symbol (e.g., `@foxandfriends`, `@1233newcastle`, `@BarackObama`).

Hashtags are topic tags created by users. By tagging one's tweet with a hashtag, users are able connect their tweets with other tweets containing the same hashtag. A user can retrieve all tweets containing a common hashtag by clicking on the hashtag. Usernames prefixed by the @ symbol are used to mention or reply to another user.

Keywords are words related to disasters. For each data instance, the *keyword* field contains a single keyword or a keyphrase found in the raw tweet. Keyphrases are stored as a single string, with "%20" replacing the space between two words (e.g., `emergency%20services`, `mass%20murder`). A small number of data instances have no keyword.

The *location* field contains the name of a location, usually a country or a city, or multiple names (e.g., `Toronto, Canada`; `Toronto, Ontario`). For some data instances, the location field contains nonsensical text (e.g., `Under Ya Skin`; `uncanny valley`). A significant proportion of data instances have no location information.

Each training data instance is labeled as 1 (the tweet is about an actual disaster) or 0 (the tweet is not about an actual disaster). This class label is stored in the *target* field. We will call a tweet in class 1 a *disaster tweet* and a tweet in class 0 a *non-disaster tweet*. The following are examples of two training data instances, one from each class:

id: `48`

keyword: `ablaze`

location: `Birmingham`

text: `@bbcmtd Wholesale Markets ablaze http://t.co/lHYXEOHY6C`

target: `0`

id: `50`

keyword: `ablaze`

location: `AFRICA`

text: `#AFRICANBAZE: Breaking news:Nigeria flag set ablaze in Aba. http://t.co/2nndBGwyEi`

target: `1`

The test dataset from Kaggle has a similar structure, except that there are no target labels. There are 3263 raw tweets in the test dataset, most of which contain words related to disasters.

We note that some data instances have missing values for the *keyword* and *location* fields. Table 2.1 shows some statistics for the training and test datasets. Table 2.2 shows the number of unique keywords, hashtags and usernames in the raw tweets.

	Number of data instances	
Statistic	**Training**	**Test**
All data instances	7613	3263
Positive class (label 1)	3271 (43.0%)	-
Negative class (label 0)	4342 (57.0%)	-
No keyword value	61 (0.8%)	26 (0.8%)
No location value	2534 (33.3%)	1106 (33.9%)

Table 2.1: Raw data statistics: target labels and missing values

Statistic	**Training**	**Test**
Number of unique keywords	222	222
Number of unique hashtags	1889	1109
Number of unique usernames	2317	1141

Table 2.2: Raw data statistics: unique keywords, hashtags and usernames

The proportion of data instances in each class is quite balanced. We would need to deal with the missing *keyword* and *location* values when we preprocess the data.

2.3 Bag of Words

In Natural Language Processing, text data needs to be tokenized and word tokens are often transformed into features. A feature could be a word, or a phrase, or a string of characters. For a dataset of millions of documents, there would be a very large number of unique text tokens. Many of these tokens are likely to be noise.

The goal of feature engineering for text data is to extract a set of tokens which are potentially important features. This will be the set of features to model the information in each data instance. This form of data representation is known as the bag of words model.

Despite the name, many of the tokens might not be valid words. It is often useful to stem all words during text preprocessing, which will truncate a word into its most basic form (e.g., `prettier` becomes `pretti`, `passenger` becomes `passen`). The stemming algorithm may also replace some characters at the end of a word (e.g., `pretty` becomes `pretti`). A token in the bag of words could also be a string of symbols, such as a URL. Thus we will refer to a token as a *term* or a *feature*, rather than a word.

2.4 Text Preprocessing

We first preprocess the training data before selecting the set of features. The test data will be preprocessed in a similar way when we prepare the test dataset for predictions.

For the raw tweets in the *text* field, we preprocess the text as follows:

- tokenize all text
- convert all characters to lower case
- remove stop words
- remove all punctuation except `#` and `@`, which represent hashtags and usernames respectively
- remove all numbers, but retain words containing numbers, which could be influential (e.g., `mh370`, the flight number of a plane that had crashed)
- stem all words
- remove all single-character terms

For the words in the *keyword* field, we preprocess the text as follows:

- tokenize all text
- convert all characters to lower case
- remove all punctuation
- remove all numbers
- append `kw_` to each keyword

There are 222 unique *keyword* tokens. Some of them are similar words in different forms, such as `bomb`, `bomb`, and `bombing`. We could stem the keywords in order to transform different forms into the same token and reduce the number of *keyword* tokens. But our experiments show that retaining the different forms results in better performance. The list of keyword tokens is shown in Appendix A.

In the training dataset, 61 data instances have missing *keyword* values. We fill in these *keyword* fields manually with one of the 222 existing *keyword* tokens, if the words are found in the raw text. For 16 of these instances whose raw text do not contain any disaster keywords, we fill the *keyword* field with `na`.

We do the same for the 26 data instances with missing *keyword* values in the test dataset. We fill 9 of the 26 instances with `na` in the *keyword* field.

About one-third of the data instances in the training and test datasets have missing *location* values. A significant number of the remaining instances have nonsensical *location* values. Therefore, we will not select any features from this field. The entire *location* column is deleted.

2.5 Feature Selection

After preprocessing of the training data, the complete set of tokens in the *keyword* and *text* fields consist of:

- 222 keywords
- 1889 hashtags prefixed by #
- 2317 usernames prefixed by @
- 15,610 other tokens, including URLS with no punctuation (e.g., `httpstcodehmym5lpk`)

To have an idea of the nature of hashtags, usernames, and other tokens with high frequencies, we generate the top 100 lists of these tokens. These lists are shown in Appendices B, C, and D.

We believe that the hashtags will be important features. A tweet containing hashtags such as `#earthquake` or `#fukushima` is likely to be about an actual disaster. Hashtags of news media companies such as `#cnn` and `#bbc` are also likely to be found in tweets about actual disasters.

Among the most frequently occurring usernames, there are celebrity usernames such as `@jimmyfallon` and `@barackobama`. Tweets containing such links are not likely to be disaster tweets. Likewise, we expect usernames of companies like `@ebay` and `@manutd` to be found in non-disaster tweets. However, we would expect disaster tweets to contain links to news media companies, such as `@reuter` and `@usatoday`.

The remaining tokens are a mix of disaster words and general words. Since all the *keyword* tokens are taken from the tweet, there is likely to be correlation between a *keyword* token and its corresponding *text* tokens.

We select a preliminary bag of words according to the following criteria:

- All *keyword* tokens
- All hashtag tokens with a minimum frequency of 2
- All username tokens with a minimum frequency of 2
- All remaining tokens with a minimum frequency of 4

Using this criteria, the resulting bag of words contain a total of 3264 features.

2.6 Feature Values

The selected bag of words is a set of categorical features which must be represented by a numerical vector. In other words, a score, or a weight, must be assigned to each feature. In general, there are three choices of weights for text features:

- Binary values (i.e., one-hot encoding)
- tf weights (term frequency in each document, or data instance)
- tf-idf weights (term frequency-inverse document frequency)

tf and tf-idf weights are useful for longer documents in which a term may occur multiple times in each document. tf-idf gives more weight to terms that occur in fewer documents. For very short documents, binary values are suitable.

A document in our dataset is a single tweet, which is no more than 140 characters. After preprocessing, the length of tweet will be shorter. Therefore we use one-hot encoding to transform the bag of words into a vector of length 3624 for each data instance. The data is very sparse, that is, in each vector, only a very small number of entries will have the value 1.

2.7 Test Dataset Preprocessing and Prediction

When we have selected the best model, we use it to predict the unknown classes in the test dataset. Before doing that, we need to preprocess the test data in the same way as the training data. Each test instance will be represented by the same set of features as the training data and one-hot encoded in the same way. After we predict the classes for all 3263 test instances, we upload our predictions to Kaggle to see our results.

2.8 Performance Metrics

Kaggle evaluates the performance using the F_1 metric, defined by:

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

where Precision is the fraction of *all predicted positive instances* that are predicted correctly:

$$\text{Precision} = \frac{\#\text{ of true positives}}{\#\text{ of true positives} + \#\text{ of false positives}}$$

and Recall is the fraction of *all positive instances in the dataset* that are predicted correctly:

$$\text{Recall} = \frac{\text{\# of true positives}}{\text{\# of true positives} + \text{\# of false negatives}}$$

During training, we also use the F_1 metric to evaluate each iteration in cross validation.

2.9 Challenges

The biggest challenge for text data is feature selection: how do we determine the best set of tokens as features? How do we measure the relevance or importance of a feature?

For this project, we have relied on raw data analysis and intuition to select a preliminary set of features. When we train our models, we can examine the learned parameters to see if there are any features with little influence and may be discarded. Some algorithm implementations provide statistics of the trained model. For example, in tree models, statistics on the number of internal nodes for each feature give an indication of the importance of the feature.

Another challenge is tuning the model. It is not easy finding the optimal set of hyperparameters, especially for complex algorithms with many hyperparameters. One solution is to use the grid search function provided in scikit-learn (`sklearn.model_selection.GridSearchCV`). For complex algorithms that are slow to train, this can be computationally costly.

Chapter 3

Experiments and Results

3.1 Algorithms

We train our training dataset of 3624 features using these algorithms:

- Logistic Regression
- Naive Bayes (Bernoulli)
- Naive Bayes (Multinomial)
- K Nearest Neighbors
- Support Vector Machine (Linear)
- Decision Tree
- Random Forest
- AdaBoost
- Gradient Tree Boosting

We use five-fold cross validation and the F_1 metric for evaluation. The following sections describe the hyperparameters we have adjusted for each algorithm to give the best result. Any other hyperparameters that are not mentioned are the default values in scikit-learn.

3.1.1 Logistic Regression

`LogisticRegression(C=0.3, max_iter=100)`

`C` is the regularization parameter which is the inverse of regularization strength. `max_iter` is the maximum number of iterations for the algorithm to converge.

3.1.2 Naive Bayes Bernoulli

`BernoulliNB(alpha=1.5)`

The Bernoulli Naive Bayes classifier is suitable for categorical features whose values are binary. `alpha` is the smoothing parameter.

3.1.3 Naive Bayes Multinomial

`MultinomialNB(alpha=35)`

The Multinomial Naive Bayes classifier is suitable for categorical features whose values are counts, such as term frequencies of documents. `alpha` is the smoothing parameter.

3.1.4 K Nearest Neighbors

`KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto')`

`n_neighbors` is the number of neighbors to use. Using a larger number of neighbors will reduce the effects of noise but in exchange, the classification boundaries are less defined. `weights` is the type of function to use for prediction. Using 'uniform' for `weights` means that all points in the neighborhood are weighted equally. The other option is to use 'distance', which means that neighbors closer to a data point have more influence compared to those further away. `Algorithm` refers to the algorithm used for calculating the nearest neighbor. Using 'auto' means it will select the most fitting algorithm based on the training data used to train the model.

3.1.5 Support Vector Machine (Linear)

`LinearSVC(C=0.01, loss='squared_hinge', max_iter=1000)`

`C` is the regularization parameter which is the inverse of regularization strength. `loss` is the loss function used. `hinge` is the standard SVM loss, while `squared_hinge` is the square of hinge loss. `max_iter` is the maximum number of iterations to run. This is important since there will be cases when the algorithm does not converge. By setting a maximum number of iterations, the algorithm will stop training whether it has converged or not.

3.1.6 Decision Tree

`DecisionTreeClassifier(max_depth=None, min_samples_split=2, min_samples_leaf=1, max_features = None)`

Decision trees are a non-parametric learning method. `max_depth` is the maximum depth of the tree. Using 'None' means the tree will keep growing until all leaves contain less samples than the number set by `min_samples_split` or until all leaves are pure. `min_samples_split` is the minimum number of samples required before splitting is allowed. `min_sample_leaf` is the minimum number of samples left at a node. Splitting will only be considered if there is a minimum number of samples in each of the left and right branches. `max_features` is the number of features to consider for the best split. Using 'none' means that the total number of all features will be used.

3.1.7 Random Forest

`RandomForestClassifier(n_estimators=500, max_depth=None, min_samples_split=2, min_samples_leaf=1, max_features='auto')`

Random Forest is an ensemble classifier that uses a combination of many decision trees. Each tree is trained by randomly sampling a subset of features.

The two most important parameters are `n_estimators` and `max_features`. `n_estimators` sets the total number of trees. Generally, a larger number of trees gives better results but it will take longer to train. `max_features` is the number of features to consider when splitting a node. The lower it is, the higher the reduction of variance, but also the greater the increase in bias. `max_depth`, `min_samples_split`, `min_samples_leaf`, and `max_features` are similar to the corresponding parameters for the Decision Tree algorithm. Setting 'auto' for `max_features` means that `max_features` is the square root of the total number of training features.

3.1.8 AdaBoost

`AdaBoostClassifier(base_estimator=None, n_estimators=50, learning_rate=1)`

AdaBoost is an ensemble boosting classifier. It works by first fitting a classifier on the dataset. It then proceeds to fit additional copies of the classifier on the same dataset, while adaptively adjusting the weights assigned to misclassified instances so that the subsequent classifier focuses more on the misclassified observations.

The two most important parameters are `n_estimators` and the choice of the `base_estimator`. `base_estimator` is the base estimator from which the ensemble is built. Setting 'none' means using the default `DecisionTreeClassifier(max_depth=1)`. `n_estimators` is the maximum number of estimators at which boosting is terminated. `learning_rate` decreases the contribution of each classifier by the value set. There is a trade-off between `learning_rate` and `n_estimators`.

3.1.9 Gradient Tree Boosting

`GradientBoostingClassifier(n_estimators=50, learning_rate=1, min_samples_split=2, min_samples_leaf=1, max_depth=3, max_features='None')`

This is the scikit-learn implementation of gradient tree boosting. The two most important parameters are `n_estimators` and `learning_rate`. `n_estimators` controls the number of weak learners while `learning_rate` controls updates of weights in each boosting round.

For `n_estimators`, a large value generally gives better performance. However, we found during tuning that a small value of 50 gave better performance for our data than other larger values. The trade off is that we need to set a larger `learning_rate`. `min_samples_split` is the minimum number of samples required before splitting is allowed. `min_sample_leaf` is the minimum number of samples left at a node. `max_depth` is the maximum depth of each tree. `max_features` is the number of features to sample for the best split. Using 'none' means that the total number of training features is used.

We will describe the algorithm in more detail in Section 3.3 on the XGBoost library.

3.2 Cross-validation Results

Table 3.1 presents the cross-validation results.

Algorithm	F_1 (%)
Logistic Regression	78.5
Naive Bayes (Bernoulli)	76.2
Naive Bayes (Multinomial)	73.9
K Nearest Neighbors	56.7
Support Vector Machine (Linear)	74.5
Decision Tree	70.5
Random Forest	73.8
AdaBoost	68.8
Gradient Tree Boosting	72.5

Table 3.1: Cross-validation F_1 results for all algorithms

3.2.1 Conclusions

From Table 3.1, we see that the Logistic Regression model has the best performance. This is not so surprising considering the fact that Logistic Regression is a simple linear model which generally works well with different types of data inputs. It is also very fast to train. Thus logistic regression is often used as a baseline model and will serve as our baseline result as well.

K Nearest Neighbors performed the worst. This is likely due to the fact that our features are one-hot encoded and the training data is high-dimensional and is extremely sparse. K Nearest Neighbor uses Euclidean distances to compute the distances between data instances, and Euclidean distance is not a good measure of similarity for sparse one-hot encoded data. For one-hot encoded data, the Euclidean distance between two instances will be the square root of a summation of 1s and 0s, that is, the value of the distance depends on the number of 1s in the summation. If the data is very sparse, it is highly unlikely for two instances to have a large number of common tokens. This implies that the distances between any two instances with the same N number of tokens (and thus will have N number of 1s in their respective vectors) would likely be the same. Conversely, the distance between one instance with a small number of tokens and another with a large number of tokens would be large, even if they are quite similar with some common tokens between them.

The ensemble models (Random Forest, AdaBoost, Gradient Tree Boosting) performed respectably. Results from other Kaggle competitions have shown that Gradient Tree Boosting is the algorithm of choice for many high performing competition entries. In particularly, the XGBoost library is widely used because it offers superior performance with regards to speed and other features such as parallelization, memory efficiency, and automatic handling of missing feature values.

3.3 XGBoost

In our remaining experiments, we will attempt to improve on our baseline result from the Logistic Regression model. We will use Gradient Tree Boosting from the XGBoost library and try to tune it to achieve better results.

Boosting is an ensemble learning method in which the algorithm adaptively assigns weights to data instances at each boosting round. Higher weights are assigned to data instances which are currently misclassified so that these instances have a higher chance of being sampled at the next round.

XGBoost is a library which implements boosting algorithms with gradient descent. The base classifier can be a linear function or a decision tree. Before deep learning methods started to outperformed traditional machine learning algorithms, gradient tree boosting models (also known as gradient boosting machines) achieved state-of-the-art results on many benchmarks [5] and was used in a number of Kaggle winning entries.

3.3.1 Model Hyperparameters

We will now use XGBoost to train a Gradient Tree Boosting model. The following are the hyperparameters which we tune for our tree models:

- `n_estimators` (range $[0,\infty]$, default = 100) Number of boosting rounds.
- `learning_rate` (range [0,1], default = 0.3) Learning rate for updating of feature weights.
- `max_depth` (range $[0,\infty]$, default = 6) Maximum depth of a tree.
- `subsample` (range [0,1], default = 1) Fraction of data instances to be randomly sampled to train each tree.
- `colsample_bytree` (range [0,1], default = 1) Fraction of features to be randomly sampled to train each tree.

3.3.2 Experiments

To determine the best hyperparameter values, we do a grid search over each parameter. The most important hyperparameters are `n_estimators` and `learning_rate`. For gradient tree boosting models, there is a trade-off between the number of trees and the learning rate [6]. A larger number of trees would require a smaller value of learning rate and vice versa.

First, we fix the `learning_rate` = 0.1 and set `colsample_bytree` = 0.5, and `subsample` = 0.5. The default value of `max_depth` = 6 is used. Using a subsample ratio of 0.5 of the training instances and features for each tree produces weaker trees and allows incremental learning. Subsampling ratios of 0.3 to 0.5 have been shown to produce better models than using the entire set of instances and features [7].

Figure 3.1 show the result of the grid search over `n_estimators`, using intervals of 100 from `n_estimators` = 100 to `n_estimators` = 2000.

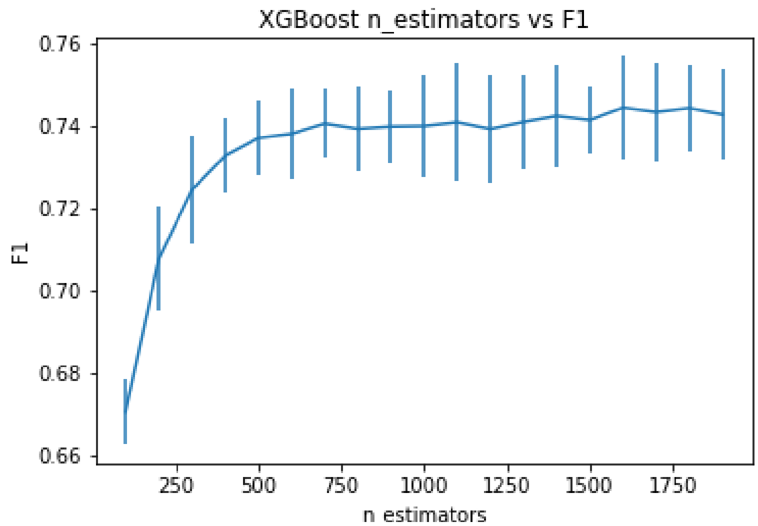

Figure 3.1: Plot of F_1 score vs `n_estimators`

We see that the F_1 score increases steadily and then plateaus from around `n_estimators` = 700 onwards. Statistics from the algorithm show that `n_estimators` = 1600 produced the best performance. However we shall use `n_estimators` = 700 to shorten the training time.

We now set `n_estimators` = 700 and do a grid search over `max_depth`. Figure 3.2 shows the result of the grid search using intervals of 1 from `max_depth` = 4 to `max_depth` = 10.

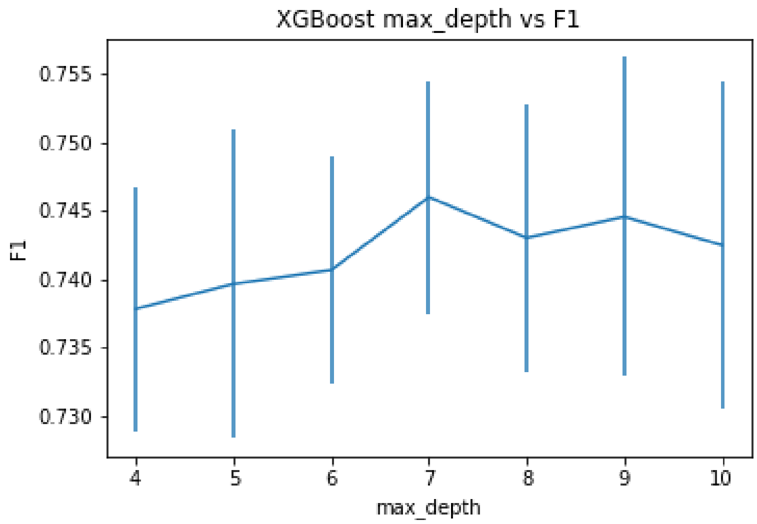

Figure 3.2: Plot of F_1 score vs `max_depth`

3.3.3 Final Model

We further tuned the `subsample` ratio and experimentated with `colsample_bylevel` (range [0,1], default = 1) and `colsample_bynode` (range [0,1], default = 1). The latter two parameters set the sampling ratio for features for each depth level and for each node respectively.

We obtain our best model using the following set of hyperparameters:

- `n_estimators` = 700
- `learning_rate` = 0.1
- `max_depth` = 7
- `subsample` = 1
- `colsample_bytree` = 0.5
- `colsample_bylevel` = 0.5
- `colsample_bynode` = 1

Table 3.2 shows the best F_1 scores for each set of parameters. `learning_rate` = 0.1 and `colsample_bynode` = 1 are fixed for all models.

Parameters	Best F_1 Score
n_estimators = 1600, max_depth = 6, subsample = 0.5, colsample_bytree = 0.5, colsample_bylevel = 1	74.5%
n_estimators = 700, max_depth = 6, subsample = 0.5, colsample_bytree = 0.5, colsample_bylevel = 1	74.6%
n_estimators = 700, max_depth = 7, subsample = 1, colsample_bytree = 0.5, colsample_bylevel = 0.5	76.2%

Table 3.2: XGBoost training results

3.4 Leaderboard Result and Conclusions

Figure 3.3 shows a screenshot of our test score on the Kaggle leaderboard. As of November 29, 2020, our position on the leaderboard was 666 out of 1269 entries. It must be noted that a majority of the top 100 scores on the leaderboard are perfect scores of 1.0, because it has been reported that the test dataset ground truths can be found online. Among the legitimate scores in the top 100 rankings, most scores are between 84% and 85%. Based on the discussion and solutions shared by other Kaggle users on the website, we believe that deep learning methods are necessary to further improve the test results. Thus our score of 79.5% shows that our Gradient Tree Boosting model is relatively successful in predicting disaster tweets. When we used our baseline Logistic Regression model to predict the test instances, the result in Kaggle was slightly worse than this best score. Considering that our test result is much better than the cross-validation training result, it shows that our Gradient Tree Boosting model generalizes well and that it is not overfitted.

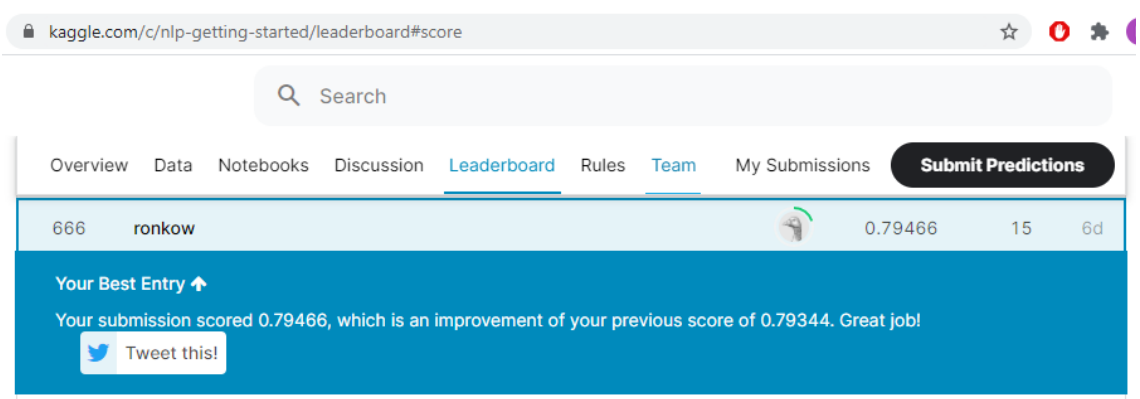

Figure 3.3: Screenshot of the Kaggle leaderboard (November 30, 2020)

Chapter 4

Conclusion

We conclude our report with an overall summary and a discussion on the lessons we have learned.

4.1 Summary

In this project, we analyzed a set of 7614 raw tweets containing words related to disasters. Our objective was to build a text classification model to classify a tweet into one of two classes: class 1 (a disaster tweet), or class 0 (a non-disaster tweet).

From the raw data, we selected a bag of words as features. This set of tokens contain 3624 tokens consisting of keywords related to disasters, hastags prefixed with the # symbol, usernames prefixed with the @ symbol, and other frequently occurring words in the tweets. The disaster keywords are provided in a separate column in the raw data. The hastags, usernames, and other words are extracted from the raw tweets after text pre-processing.

We experimented with various basic and ensemble classification algorithms. Each model is trained using five-fold cross validation and evaluated using the F_1 metric. The logistic regression model produced the best performance, with an cross-validation score of 78.5%. Next, we experimented with the XGBoost library. Our best performing model is a Gradient Tree Boosting model, trained using XGBoost, which achieves a F1 score of 79.5% on the official test dataset in Kaggle. Although the baseline model has a higher training score than the Gradient Tree Boosting model, its test result is slightly worse.

Based on the discussion and solutions shared by other Kaggle users on the website, we believe that deep learning methods are necessary to improve the test results. Most of the top 100 F1 scores on the leaderboard are between 84% and 85%. Thus our score of 79.5% shows that our Gradient

Tree Boosting model is relatively successful in predicting disaster tweets.

4.2 Lessons Learned

We learned some valuable lessons from working on this project. First, we found that using a more complex algorithm does not necessarily give better results. Logistic Regression outperformed more complicate ensemble algorithms in our experiments.

Second, when tuning an algorithm that performs poorly, it is hard to tell whether it is due to poor choice of hyperparameter values or due to the fact that the algorithm does not work well with the training data. Some algorithms might not give improved performance no matter how the parameters are tuned. We can only conclude that the problem lies in the algorithm after much tuning. While this is part and parcel of experimentation in machine learning, it is time consuming and costly.

Our conclusion is that the performance of a model depends on the type and nature of the training data (binary or continuous values, sparse data, etc.), the size of the training dataset, and the type of classification task (binary or multi-class).

4.3 Kaggle Competition Page and Source Files

The Kaggle webpage for this project is at:

`https://www.kaggle.com/c/nlp-getting-started/`

The source files, including the data, models, and results, are accessible at:

`https://github.com/ronkow/kaggle-disaster-tweets`

4.4 Allocation of Work

Work was allocated as follows:

- Data preprocessing: Ron
- Model training: Brandon, Ron
- Report writing: Brandon, Ron

Bibliography

[1] Kaggle. https://www.kaggle.com/. 2

[2] Twitter. https://twitter.com/. 1

[3] Weibo. https://weibo.com/. 1

[4] Xgboost. https://xgboost.ai/. 4

[5] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016. 4, 16

[6] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001. 16

[7] Jerome H Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002. 16

Appendices

A List of 222 Keyword Tokens

kw_ablaze
kw_accident
kw_aftershock
kw_airplaneaccident
kw_ambulance
kw_annihilated
kw_annihilation
kw_apocalypse
kw_armageddon
kw_army
kw_arson
kw_arsonist
kw_attack
kw_attacked
kw_avalanche
kw_battle
kw_bioterror
kw_bioterrorism
kw_blaze
kw_blazing
kw_bleeding
kw_blewup
kw_blight
kw_blizzard
kw_blood
kw_bloody
kw_blownup
kw_bodybag
kw_bodybagging
kw_bodybags
kw_bomb
kw_bombed
kw_bombing
kw_bridgecollapse
kw_buildingsburning
kw_buildingsonfire
kw_burned
kw_burning
kw_burningbuildings
kw_bushfires
kw_casualties
kw_casualty
kw_catastrophe
kw_catastrophic
kw_chemicalemergency
kw_clifffall
kw_collapse
kw_collapsed
kw_collide
kw_collided
kw_collision
kw_crash
kw_crashed
kw_crush
kw_crushed
kw_curfew
kw_cyclone
kw_damage
kw_danger
kw_dead
kw_death
kw_deaths
kw_debris
kw_deluge
kw_deluged
kw_demolish
kw_demolished
kw_demolition
kw_derail
kw_derailed
kw_derailment
kw_desolate
kw_desolation
kw_destroy
kw_destroyed
kw_destruction
kw_detonate
kw_detonation
kw_devastated
kw_devastation
kw_disaster
kw_displaced
kw_drought
kw_drown
kw_drowned
kw_drowning
kw_duststorm
kw_earthquake
kw_electrocute
kw_electrocuted
kw_emergency
kw_emergencyplan
kw_emergencyservices
kw_engulfed
kw_epicentre
kw_evacuate
kw_evacuated
kw_evacuation
kw_explode
kw_exploded
kw_explosion
kw_eyewitness
kw_famine
kw_fatal
kw_fatalities
kw_fatality
kw_fear
kw_fire
kw_firetruck
kw_firstresponders
kw_flames
kw_flattened
kw_flood
kw_flooding
kw_floods
kw_forestfire
kw_forestfires
kw_hail
kw_hailstorm
kw_harm
kw_hazard
kw_hazardous
kw_heatwave
kw_hellfire
kw_hijack
kw_hijacker
kw_hijacking
kw_hostage
kw_hostages
kw_hurricane
kw_injured
kw_injuries
kw_injury
kw_inundated
kw_inundation
kw_landslide
kw_lava
kw_lightning

kw_loudbang
kw_massacre
kw_massmurder
kw_massmurderer
kw_mayhem
kw_meltdown
kw_military
kw_mudslide
kw_na
kw_naturaldisaster
kw_nucleardisaster
kw_nuclearreactor
kw_obliterate
kw_obliterated
kw_obliteration
kw_oilspill
kw_outbreak
kw_pandemonium
kw_panic
kw_panicking
kw_police
kw_quarantine
kw_quarantined
kw_radiationemergency
kw_rainstorm
kw_razed
kw_refugees
kw_rescue
kw_rescued
kw_rescuers
kw_riot
kw_rioting
kw_rubble
kw_ruin
kw_sandstorm
kw_screamed
kw_screaming
kw_screams
kw_seismic
kw_sinkhole
kw_sinking
kw_siren
kw_sirens
kw_smoke
kw_snowstorm
kw_storm
kw_tornado
kw_stretcher
kw_structuralfailure
kw_suicidebomb
kw_suicidebomber
kw_suicidebombing
kw_sunk
kw_survive
kw_survived
kw_survivors
kw_terrorism
kw_terrorist
kw_threat
kw_thunder
kw_thunderstorm
kw_tragedy
kw_trapped
kw_trauma
kw_traumatised
kw_trouble
kw_tsunami
kw_twister
kw_typhoon
kw_upheaval
kw_violentstorm
kw_volcano
kw_warzone
kw_weapon
kw_weapons
kw_whirlwind
kw_wildfire
kw_wildfires
kw_windstorm
kw_wounded
kw_wounds
kw_wreck
kw_wreckage
kw_wrecked

B List of Top 100 Hashtags and Their Frequencies (Training)

#new	73
#hot	30
#prebreak	30
#best	30
#job	26
#nowplay	24
#islam	24
#hiroshima	21
#earthquak	19
#gbbo	18
#wildfir	13
#isi	12
#terror	12
#flood	11
#world	11
#hire	11
#japan	10
#india	10
#sismo	10
#yyc	10
#break	9
#rt	9
#worldnew	9
#direction	9
#irand	9
#fashion	9
#art	9
#emmerdal	9
#cnn	9
#abstorm	9
#bbc	9
#nuclear	9
#disast	8
#edm	8
#dnb	8
#beyhiv	8
#handbag	8
#seattl	8
#genocid	8
#nurs	8
#dubstep	7
#trapmus	7
#danc	7
#u	7
#tcot	7
#bioterror	7
#busi	7
#mh370	7
#wx	7
#okwx	7
#animalrescu	7
#truth	7
#god	7
#quran	7
#li	7
#armageddon	6
#mtvhottest	6
#refuge	6
#kindl	6
#women	6
#usa	6
#tech	6
#cours	6
#photographi	6
#rohingya	6
#wmata	6
#tubestrik	6
#soundcloud	6
#militari	6
#drought	6
#prophetmuhammad	6
#quot	6
#hailstorm	6
#africa	6
#antioch	6
#fukushima	6
#bestnaijamad	6
#california	5
#lgbt	5
#tbt	5
#mumbai	5
#bb17	5
#iphon	5
#prepared	5
#technolog	5
#uk	5
#np	5
#summerf	5
#phoenix	5
#1	5
#myanmar	5
#sittw	5
#strategicpati	5
#libya	5
#usg	5
#emerg	5
#scienc	5
#allah	5
#socialnew	5
#afterlif	5

C List of Top 100 Usernames and Their Frequencies (Training)

@youtub	83	@refuge	3
@arianagrand	11	@tflbusalert	3
@potu	9	@mnpdnashvil	3
@chang	9	@ianhellfir	3
@usatoday	9	@barackobama	3
@foxnew	9	@busi	3
@emmerdal	8	@worldnetdaili	3
@djicemoon	7	@wire	3
@justinbieb	7	@zakbagan	3
@mikeparractor	6	@nickcannon	3
@stretcher	6	@realmandyrain	3
@towel	6	@itunesmus	3
@viralspel	5	@itun	3
@usagov	5	@foxysiren	3
@youngheroesid	5	@witter	3
@invalid	5	@nasahurrican	3
@ap	4	@dannyonpc	3
@localarsonist	4	@aftershockdelo	2
@lonewolffur	4	@fouseytub	2
@kurtschlicht	4	@tomcatart	2
@michael5so	4	@thisizbwright	2
@reuter	4	@rohnertparkdp	2
@gop	4	@theadvocatemag	2
@realdonaldtrump	4	@arsonistmus	2
@unsuckdcmetro	4	@diamorfiend	2
@raynbowaffair	4	@casperrmg	2
@diamondkesawn	4	@envw98	2
@graze	4	@nickcocofre	2
@rexyy	4	@juliedicaro	2
@trubgm	3	@jdabe80	2
@ebay	3	@cbsbigbroth	2
@blizzarddraco	3	@fedex	2
@nbcnew	3	@scoopit	2
@post	3	@cdcgov	2
@calum5so	3	@anellatulip	2
@nytim	3	@manutd	2
@weathernetwork	3	@meekmil	2
@sharethi	3	@drake	2
@guardian	3	@scottwalk	2
@jimmyfallon	3	@abubaraa1	2
@claytonbry	3	@amazon	2
@accionempresa	3	@teamstream	2
@gerenciatodo	3	@prosyn	2
@davidvonderhaar	3	@ameenshaikh3	2
@yahoonew	3	@themagickidrap	2
@tinyjecht	3	@minimehh	2
@spinningbot	3	@cjoyner	2
@apollobrown	3	@fewmoretweet	2
@un	3	@rockbottomradfm	2
@smh	3	@wwwcbplawyer	2

D List of Top 100 Other Tokens and Their Frequencies (Training)

like	407
fire	355
get	311
amp	301
bomb	228
new	224
via	220
one	205
go	198
peopl	197
burn	178
kill	175
video	170
time	161
crash	160
emerg	158
us	157
flood	156
attack	153
build	152
bodi	150
year	148
disast	148
look	144
say	142
polic	139
fatal	138
home	137
day	136
would	136
famili	130
love	129
make	129
still	129
evacu	128
see	128
train	128
come	125
storm	123
know	122
back	121
watch	118
want	117
suicid	117
news	117
california	116
live	115
bag	115
scream	114
collaps	114
derail	114
got	112
car	112
man	110
death	110
first	107
take	105
think	105
world	105
caus	104
today	101
need	101
work	100
drown	100
two	99
rt	99
wreck	99
let	98
war	97
dead	96
deton	96
destroy	95
accid	94
plan	93
feel	93
nuclear	92
hijack	92
fuck	91
full	91
fear	90
obliter	90
good	89
may	88
murder	88
weapon	88
way	86
last	86
help	86
even	86
surviv	86
wound	85
life	84
mani	84
could	83
servic	82
injuri	81
use	80
report	80
rescu	78
explod	78