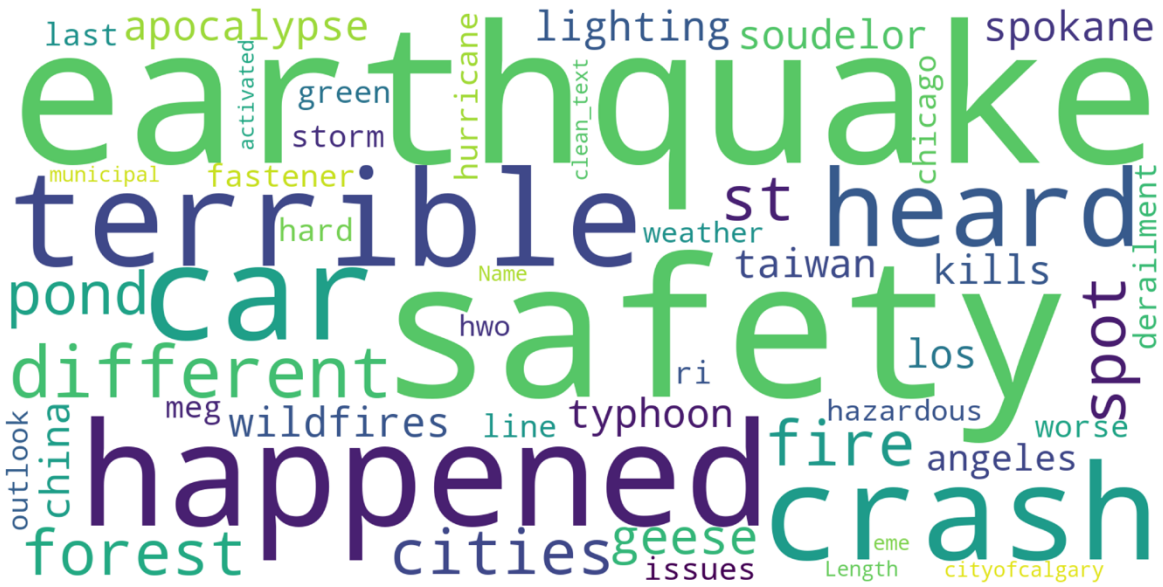
Real or Not? NLP with Disaster Tweets



by Jonas Hellevang

Python Notebook: https://github.com/Jonashellevang/MBD-JH/blob/master/Jonas_Hellevang_NLP_with_Disaster_Tweets_Kaggle.ipynb

Structure of the Python Notebook

As a main structure and process of the notebook, I started with uploading the dataset and importing libraries, followed by exploratory data analysis (EDA), a baseline using Naïve Bayes, and running different models. I chose not to do a lot of preprocessing before I introduced the baseline because I wanted to see each individuals effect on its own on the models I ran. I ran the Naïve Bayes model for every new introduced change in the CountVectorizer(), and later on when I was satisfied with my model, I ran Logistic Regression and Support Vector Machine.

Introduction and Exploratory Data Analysis (EDA)

On a general note, this assignment was in particular interesting because even though concepts and theory from class was understood, we still had not run any sort of machine learning process like this. It was not until after the process of EDA I understood that when we vectorize the text, creating new features and doing "normal" machine learning processes are completely different. For me, that was the biggest discovery of this assignment. The machine learning process is way different, and that highlights the importance of proper cleaning and processing of the text.

Datasets

For this assignment we were given one dataset for training and one dataset for testing. These were both provided on the Kaggle competition website, where we also uploaded our models for scoring. I named my models train_df and test_df respectfully and ran my EDA on train_df. The train dataset contains 7,613 tweets, while the test set has 3,263 which is not a whole lot to use for the classification.

Example Tweets

Before we really start on the EDA, it is nice to see examples of how disaster and non-disaster tweets look like, as well as what types of words and symbols that are used in the different tweets.

```
1 # Example of disaster tweet
2 print(train_df[train_df["target"] == 1]["text"].values[6])
3 print(train_df[train_df["target"] == 1]["text"].values[30])
```

```
 #flood #disaster Heavy rain causes flash flooding of streets in M
anitou, Colorado Springs areas
accident Reported motor vehicle accident in Curry on Herman Rd near
Stephenson involving an overturned vehicle. Please use... http://t.
co/YbJezKuRW1
```

```
1 # Example of a non-disaster tweet
2 print(train_df[train_df["target"] == 0]["text"].values[6])
3 print(train_df[train_df["target"] == 0]["text"].values[60])
```

```
 London is cool ;)
aftershock Aftershock was the most terrifying best roller coaster
I've ever been on. *DISCLAIMER* I've been on very few.
```

Moving from there, I checked what information there was to give about the different features:

```
1 train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7613 entries, 0 to 7612
Data columns (total 5 columns):
id          7613 non-null int64
keyword     7552 non-null object
location    5080 non-null object
text        7613 non-null object
target      7613 non-null int64
dtypes: int64(2), object(3)
memory usage: 297.5+ KB
```

The dataset has five different columns:
id: a unique identifier for each tweet
keyword: a particular keyword from the tweet
location: the location the tweet was sent from
text: the text of the tweet
targets: in train.csv only, this denotes whether a tweet is about a real disaster (1) or not (0)

keyword and location

I found the location and keyword features particularly interesting because these features could potentially tell us a lot about the different tweets. Therefore, getting more information from these features makes sense:

```
1 train_df['keyword'].nunique()
```

```
221
```

```
1 train_df['location'].nunique()
```

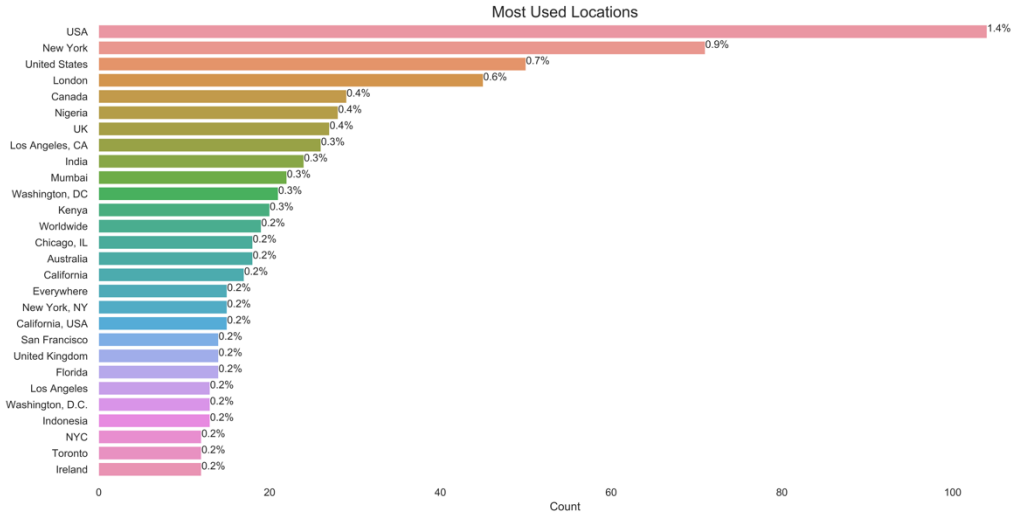
```
3341
```

```
1 # Check for NA's
2 train_df.isnull().sum()
```

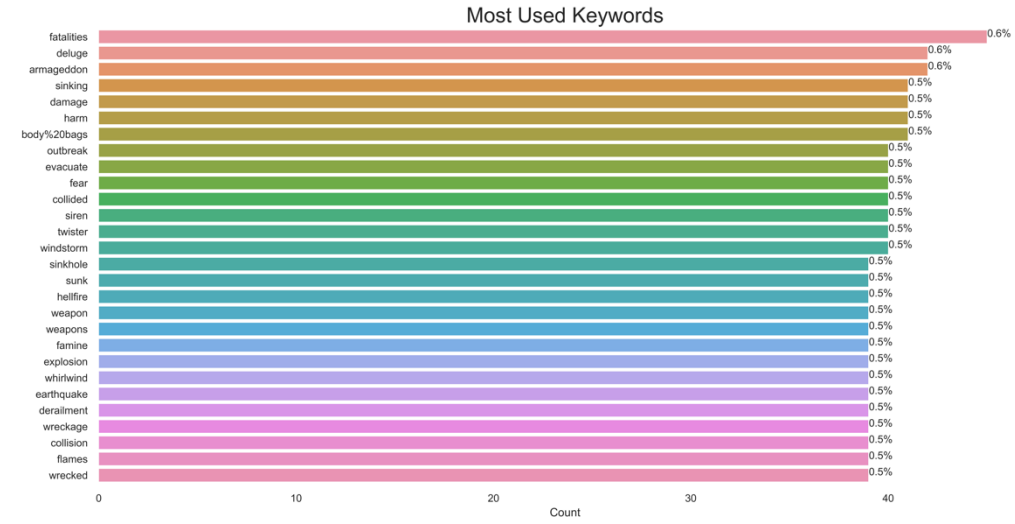
```
id             0
keyword       61
location    2533
text           0
target         0
dtype: int64
```

When running these three lines we can see that keyword and location is the only features with missing values, which makes sense since id, target and text needs values in all its rows. We can also see that keyword has 221 distinct values, and location has 3,341 distinct values. Since there are so many different values, I decided to only show the top 28 locations and

keywords. It can seem like a very random number, but by doing so I included all values that was repeated the same amount of times and did not cut the process off showing only half of the same values with the same count. The distribution can be seen in these two graphs:



As there are 3,341 unique values for location, I chose not to do any processing of this information. Even though I would like to change for example "United States" to "USA" and city names to states or something similar, it is a huge manual process that will take too long to work through. Also, the test set might have different locations that are not introduced in the train set, making it potentially useless in a machine learning process.



When looking at the graph we can see that all the words are potentially connected with a disaster tweet. Therefore, it is of high importance to include this into our vector. There are two ways of including these features. It is possible to create one classifier for the text column and one for the additional columns, followed by assembling both into the same classifier. Alternatively, like I did, it is possible to combine these keywords into the text column before vectorizing. I first tried to include both location and keyword into the vectorizer, soon realizing that location gave a worse score than keyword alone, which is why I chose to only include the keyword feature in the vectorizer.

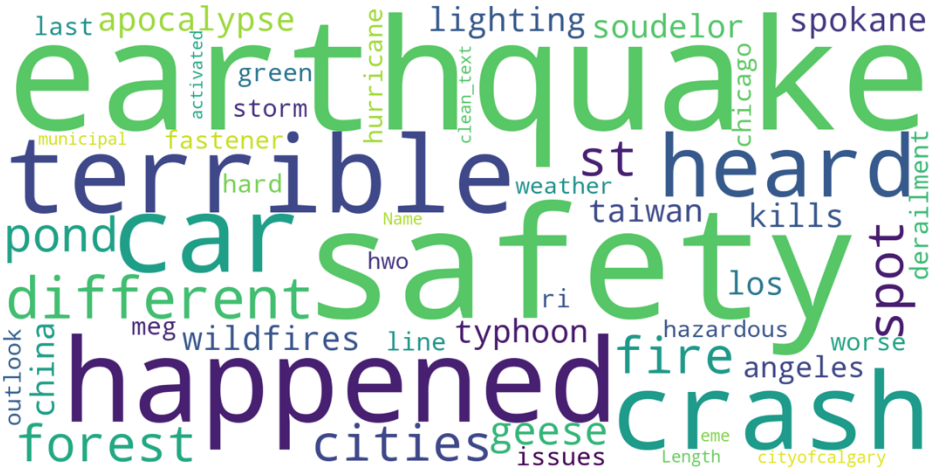
Most and Least Common Words

When running my code on most and least common words it is easy to see that the biggest issue here is the stop words. Stop words are words that are very common in our language, and you see examples of them in the most common words table below:

```
The most common words are:
 the      2575
a         1845
to        1805
in        1757
of        1722
and       1302
I         1197
for        820
is         814
on         773
-          763
you        632
The        552
my         549
with       508
that       492
at         485
by         469
it         433
from       372
dtype: int64
```

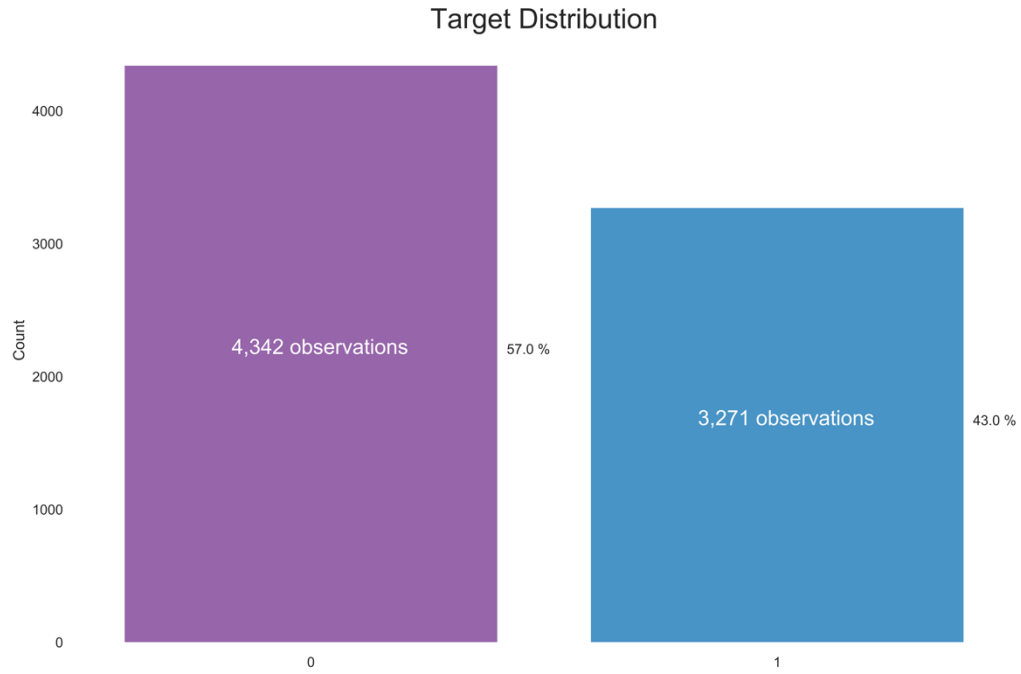
```
The least common words are:
 .....                    1
'Become                   1
http://t.co/obfD7e4QcP    1
insult                    1
convection                1
Shooting                  1
@RyleeDowns02             1
Wolforth                  1
usa                       1
Cocker                    1
Let  Ûªs                  1
http://t.co/IUh718KCy0    1
Dm                        1
http://t.co/J6mpdsx9Lk    1
Audi                      1
Effects)                  1
(N).                      1
CH4                       1
MwjCdk                    1
CATASTROPHIC              1
dtype: int64
```

I also created an illustration, which you have already seen on the front page of this report, of the most used words using the WordCloud library. This illustration is run after combining keyword with text, and removes the stop words which is why the result is so different than what the table above shows:



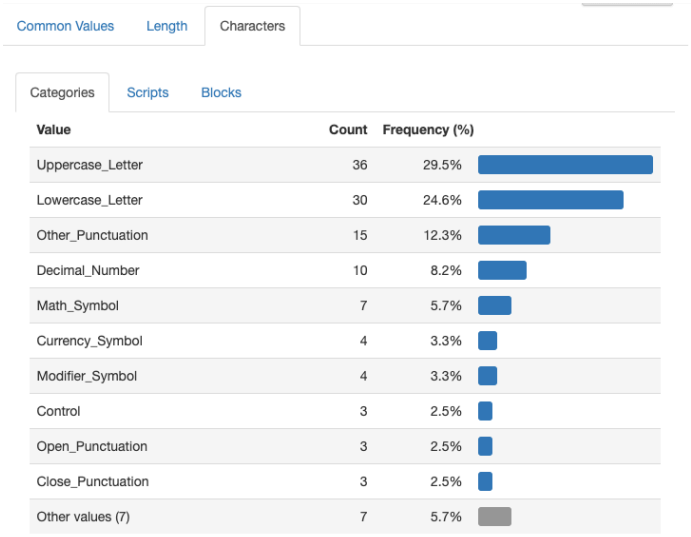
Target Distribution

When looking at the target, which is what we will train our model after, we can see that it is an uneven distribution. There are 4,342 observations of tweets that are not disaster tweets (0), and 3,271 observations of tweets that are disaster tweets. These numbers add up to 57% and 43% respectfully, which is a little harder to train than observations of equal distribution. However, the distribution of the target is not big enough for us to call it unbalanced. Therefore, we do not have, nor should we, do any changed to this distribution.



Profiling Report

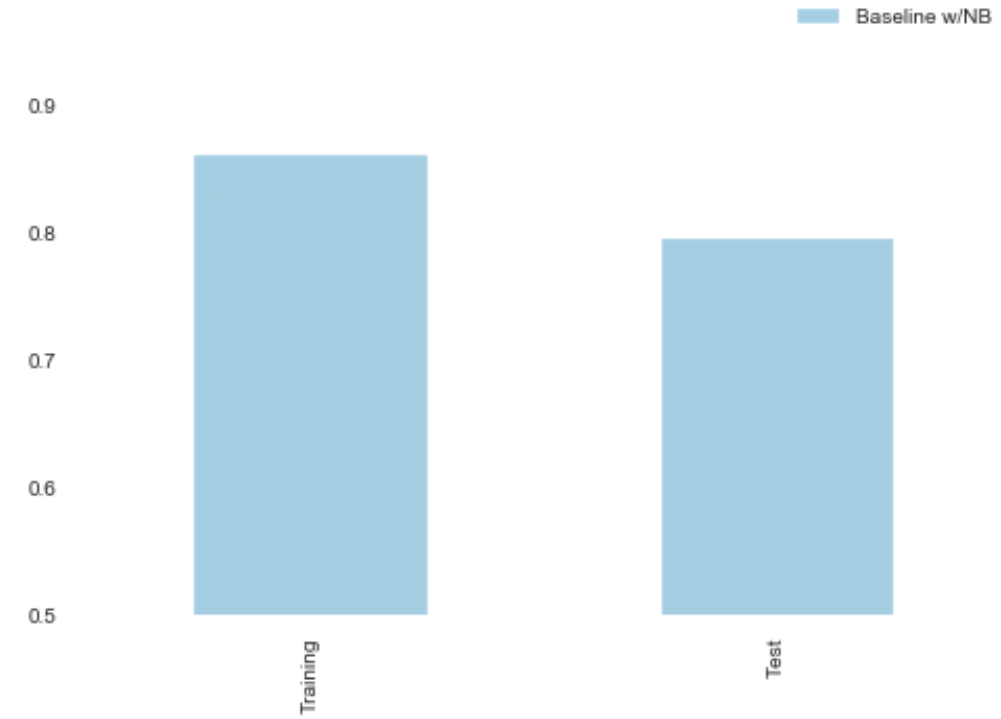
After doing the manual EDA I ran a code to get a profile report (notice that this only shows in the Jupyter Notebook if you run the code yourself). This report gives you pretty much everything you need for EDA, with a few exceptions of course. It is divided into overview, variables, interactions, correlations, missing values and sample of the data. In the overview we can see the variable types, duplicates, missing cells, number of observations, number of variables and more. We also get four warnings in our train set notifying us about high cardinality in keyword, location and text, as well as there are 33.3 % missing values in the location feature. When moving to the variables there is a lot of information we can play with, and something that is very nice to know about is how we can "toggle" the details of a feature. For example, with the feature "text", we can see that the max number of words is 164, and the mean 110. We can also get more information about the characters used, where we get this result:



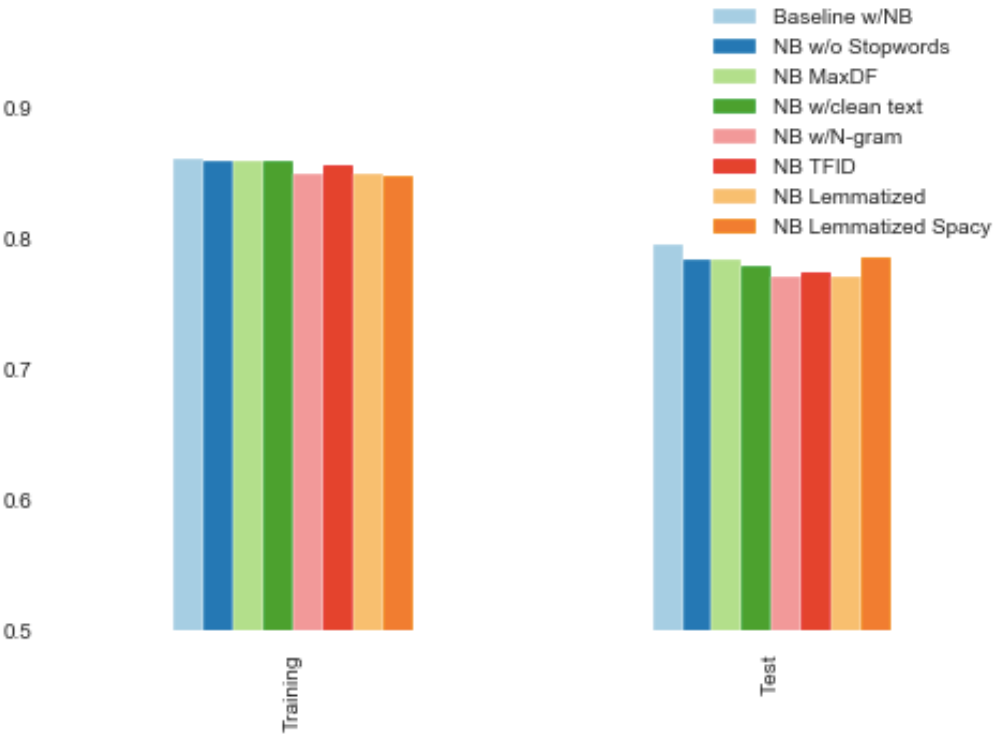
Baseline and Naïve Bayes

Baseline with Naïve Bayes

In the baseline prediction nothing was cleaned or fixed. It was achieved using Naïve Bayes, which is a machine learning model that all features, or in this case vectors, are independent of each other. The baseline had the following result:



This first model had a surprisingly high score on Kaggle of 0.79243. From there on I introduced several models using Naïve Bayes. Below image is a summary of all of the Naïve Bayes models I ran, and on the next page I will explain the different effects of them:



Naïve Bayes without stop words

Stop words is the most common words in a language, and you can see in the table where I list the most common words that they show up quite frequently compared to other words. This is exactly why we remove them, because they do not give us any information about the tweets we are analyzing.

Naïve Bayes with MaxDF

MaxDF ignore terms that have a document frequency strictly higher than the given threshold. I tried MinDF as well for this as well, but this gave an error that I chose to leave in the notebook for reference. The error said there was no words left after tuning it to "2", which in turns says only to keep words that are mentioned twice. It is not a big surprise that this did not work as it is not big documents of text.

Naïve Bayes with cleaned text

Regular expressions, or what I have called preprocessor in my Jupyter Notebook, is sets of characters or patterns. In our case, we do want to remove URLs, punctuations, emojis, "hitting enter", numbers, phone-numbers etc. This model is a result of that. To avoid confusion, the "clean_text" column was not used here as it was included in the CountVectorizer().

Naïve Bayes with N-grams

As you can see from the Jupyter Notebook, I used N-gram of 3, creating bigrams and trigrams. This was a result of some research and testing different models, which is why I chose this parameter for the CountVectorizer().

Naïve Bayes using TF-IDF Weighting

TF-IDF is short for term frequency-inverse document frequency. It is a numerical statistic that gives us information about how important a word is. With the model I created TF-IDF gave a higher result after introducing N-grams, however, the Kaggle score was too low.

Naïve Bayes Lemmatized with nltk

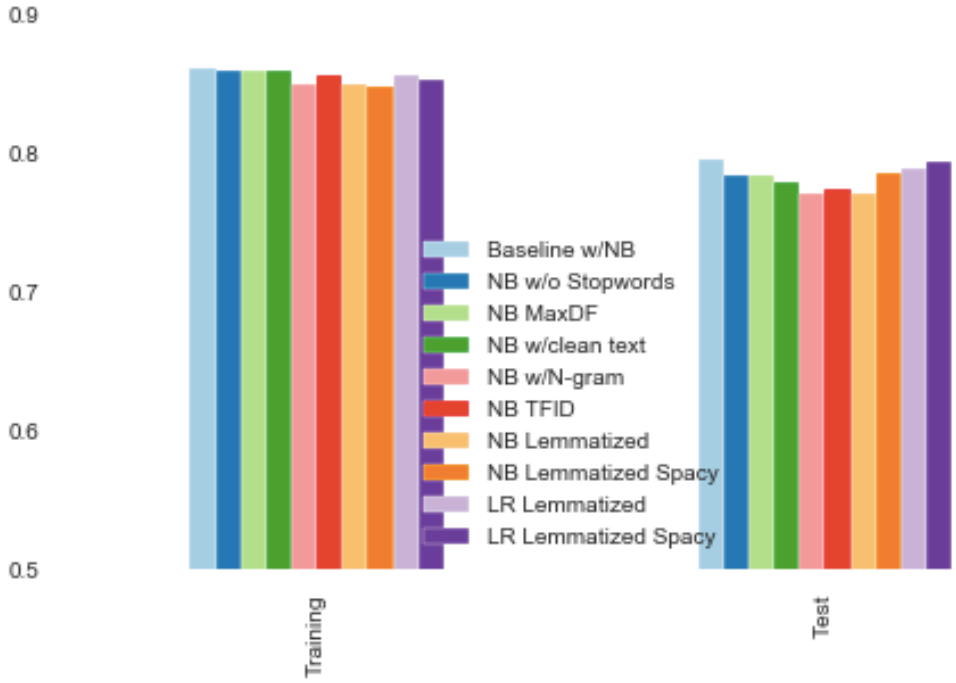
The lemmatizer shortens all the word. It does the same as stemming but adds more context to it. Here I use the standard of nltk with WordNetLemmatizer().

Naïve Bayes Lemmatized with spacy

Here the model is created with spacy as a lemmatizer. This is because nltk is a little bit outdated, which is why I wanted to check the result of implementing this in the model instead of the default of nltk.

Logistic Regression

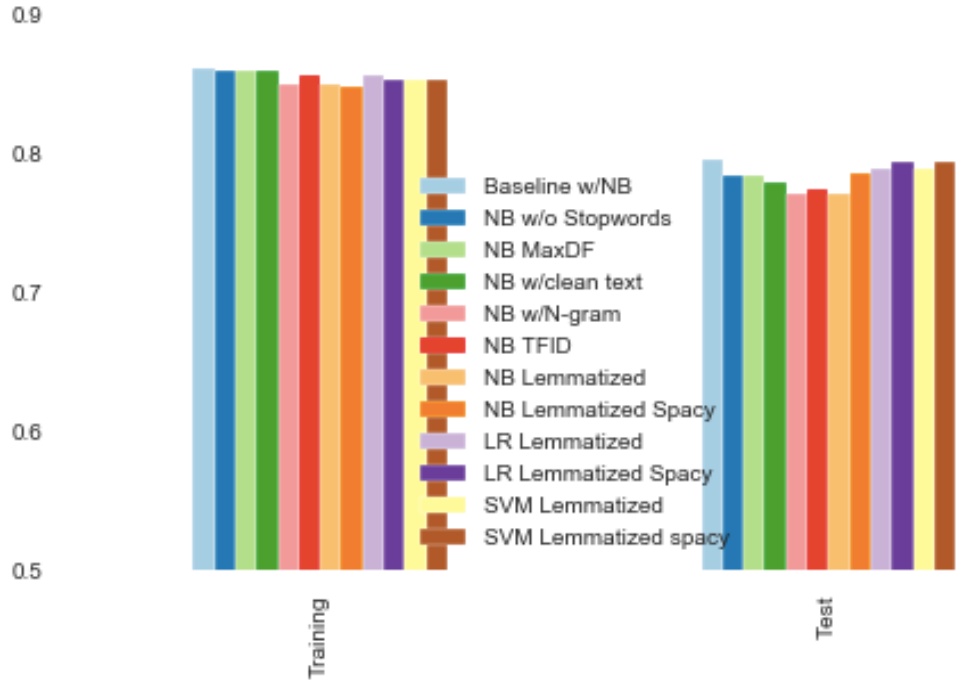
Below is the result from using logistic regression:



I created two models using regression, where I tried both with nltk lemmatizer and spacy lemmatizer. Logistic regression using nltk's WordNetLemmatizer() gave the best score so far with a score of 0.8047.

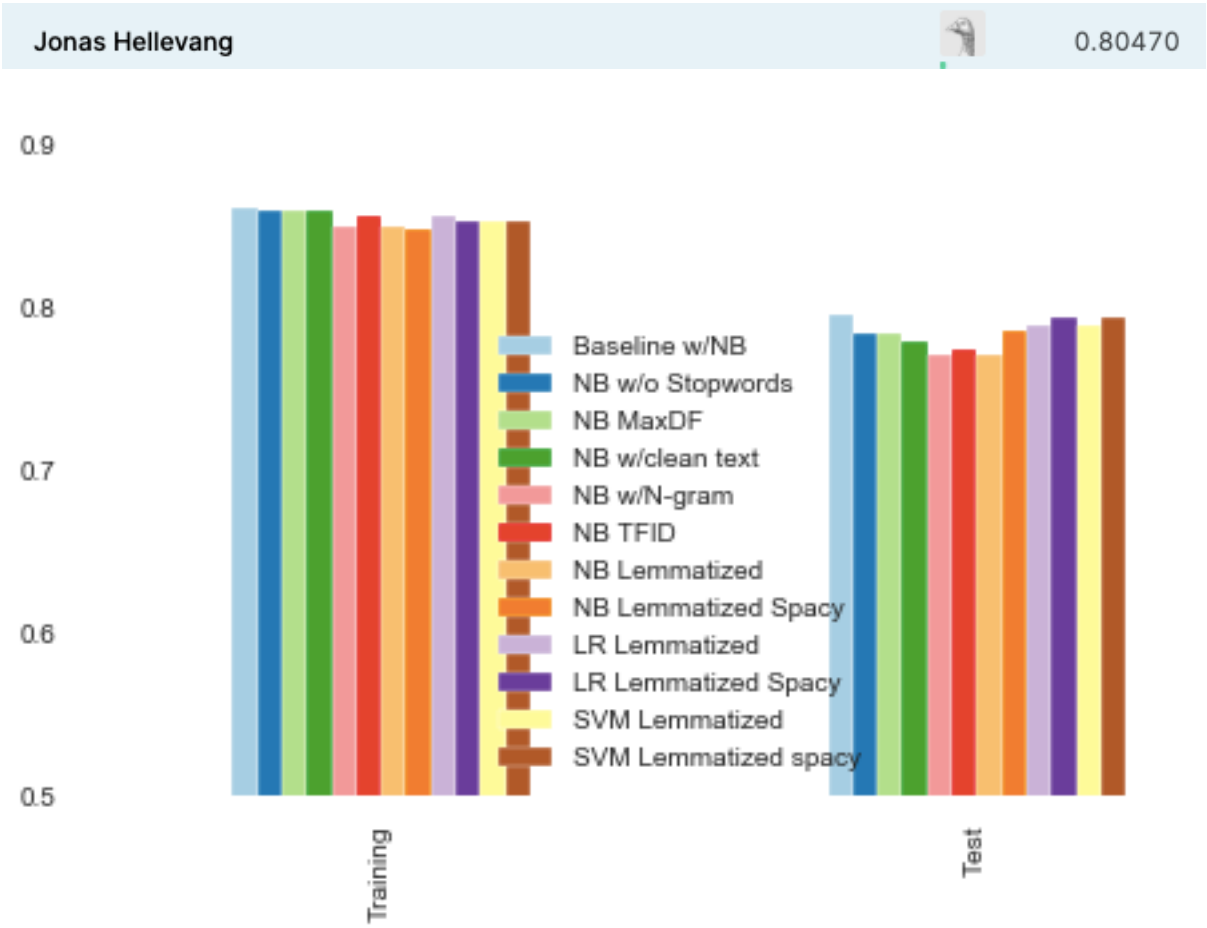
Support Vector Machine

Support vector machines finds an optimal boundary between possible outputs, but did not give the best score at all as seen in the below graph:



Conclusion and Final Model

The best model I was able to create by not creating new features was logistic regression with nltk lemmatizer. This model produced a score of 0.80470.



Possible Improvements

If I had more time, I would try how normalization works for spelling mistakes and changing for example U.S.A to USA. I would also have tried to create new variables into a vector, as in our case, the length of the tweet and length of words in tweets might have something to say for the prediction. I also wish I had more time to analyze the N-grams more than just finding the best model while using them.