Tweet Classifier: Detecting Disasters

Brandon Arcilla

Problem

The beauty of Twitter is that it offers real-time communication of users. And when a disaster happens, being able to analyze tweets can better assist in responding to the situation. That's why it's important for media outlets and disaster relief organizations, like FEMA, to monitor social media. Programming a machine learning model to do this can be difficult. There's certain words that can mean different things that humans can decipher quicker than a machine. For example, "on fire" can refer to an actual fire, as in "that building is on fire", or a term used when you eat something spicy, as in "my mouth is on fire". I believe that there are certain elements in a tweet that a model can recognize to predict whether a tweet is referring to a real disaster, like unique keywords and the length of tweets. The goal of this project is to build a model that can recognize the difference and predict what's referring to a disaster and what is not.

Data Collection, Wrangling, and Cleaning

Data is coming from Data for Everyone (https://www.figure-eight.com/data-for-everyone/) where there's a data set for about 10,000 tweets that include the tweet and whether it's relevant or not relevant to the topic of a real disaster. Since this was already available through CSV, there was not any wrangling that I needed to do.

The dataset originally had 13 columns with 10,876 entries.

```
tweets.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10876 entries, 0 to 10875
Data columns (total 13 columns):
_unit_id                 10876 non-null int64
_golden                  10876 non-null bool
_unit_state              10876 non-null object
_trusted_judgments       10876 non-null int64
_last_judgment_at        10792 non-null object
choose_one               10876 non-null object
choose_one:confidence    10876 non-null float64
choose_one_gold          87 non-null object
keyword                  10789 non-null object
location                 7238 non-null object
text                     10876 non-null object
tweetid                  10876 non-null float64
userid                   10789 non-null float64
dtypes: bool(1), float64(3), int64(2), object(7)
memory usage: 1.0+ MB
```

Out of 13 columns, I only used 2 of the columns: choose_one, which represents whether the text is Relevant, Not Relevant, or Can't Decide on the topic of disasters, and text, which is the text of tweets.

```
tweets = tweets[['choose_one', 'text']]

tweets.head(5)
```

	choose_one	text
0	Relevant	Just happened a terrible car crash
1	Relevant	Our Deeds are the Reason of this #earthquake M...
2	Relevant	Heard about #earthquake is different cities, s...
3	Relevant	there is a forest fire at spot pond, geese are...

```
tweets.groupby('choose_one').count()
```

choose_one	text
Can't Decide	16
Not Relevant	6187
Relevant	4673

The dataset has more tweets that are not related to disasters than those tweets that are related. There's also another value "Can't Decide" for tweets that could not be categorized. Since it's a small amount of the data I removed it, which allowed our target to have 2 classifications: Relevant and Not Relevant, which can then be mapped to "Yes" (Relevant) and "No" (Not Relevant).

```
df = pd.DataFrame() #Create blank dataframe

#Add new columns with data while dropping rows that are marked as "Can't Decide"
df[['target', 'text']] = tweets[tweets['choose_one'] != "Can't Decide"]
```

```
df.groupby('target').count()
```

target	text
Not Relevant	6187
Relevant	4673

```
df['target'] = df['target'].map({'Relevant': 'Yes', 'Not Relevant': 'No'})
```

```
df.head()
```

	target	text
0	Yes	Just happened a terrible car crash
1	Yes	Our Deeds are the Reason of this #earthquake M...
2	Yes	Heard about #earthquake is different cities, s...
3	Yes	there is a forest fire at spot pond, geese are...
4	Yes	Forest fire near La Ronge Sask. Canada

Before starting EDA and applying any machine learning models, the text column needed to be cleaned by removing common characters that are seen in tweets such as hashtags, mentions, and urls.

```
'When carelessness leads to an aviation accident the victim has the right to seek compensation for damages. http://t.
co/eqAG6rz1vO'
```

Along with this, the words in the text all needed to be in lowercase. Doing this will ensure that our model doesn't count 'Hello' and 'hello' as 2 separate words. I then lemmatized each word to their lemma, which shortened each word to their base; for example, lemmatization would shorten 'caresses' to 'caress' or 'dogs' to 'dog'. Finally, stop words were removed from each text. Stop words are common words in a language, such as 'The', 'But', and 'To', that have a high frequency. These stop words are not helpful for our model and having them take up space in our dataset that our model trains on. I used the NLTK stopword list and added some acronyms that we might encounter specifically in tweets; such as, rt for retweets, cc for carbon copy and more. (https://mashable.com/2013/07/19/twitter-lingo-guide/)

```
#Remove stopwords
stop_words = stopwords.words('english')

#New words to add to the stopwords list. This contains
newWords = ['afaik', 'cc', 'cx', 'dm', 'ff', 'ht', 'icymi',
            'mm', 'mt', 'nsfw', 'oh', 'prt', 'rlrt', 'rt',
            'smh', 'tftf', 'til', 'tl', 'dr', 'tmb', 'tqrt', 'tt', 'w']
stop_words.extend(newWords)

def text_cleaning(text):
    """If applying on DataFrame column, use within an apply method.
    INPUT:
        - text: text string"""

    text = re.sub(r'(www|http)\S+','',text) #Removes URLs
    text = re.sub(r'@\w+','', text) #Removes nametags
    text = text.lower() #lowercase all words

    def lemmatize(text):
        lemmatizer = WordNetLemmatizer()
        lemmatized_output = ' '.join(lemmatizer.lemmatize(x, 'v') for x in word_tokenize(text))
        return lemmatized_output

    text = lemmatize(text)

    text = re.sub(r'[^a-z\s]',' ', text) #remove random symbols and numbers in string

    def remove_stopwords(text):
        token = word_tokenize(text)

        remove_short = [x for x in token if len(x)>2 ] #remove words that are shorter than 2 letters

        #remove stopwords and put sentence back together
        remove_output = ' '.join(x for x in remove_short if x not in stop_words)

        return remove_output

    text = remove_stopwords(text)

    return text
```

This function yields a cleaned text that can now be used in analysis.

```
#Test out the text cleaning function

clean2 = df.text[7958]
print('Original:',clean2)
print()
print('Cleaned:', text_cleaning(clean2))
```

```
Original: Landslide caused by severe rainstorm kills 3 in ItalianåËAlps https://t.co/8BhvxX2Xl9 http://t.co/4ou8s82Hx
J

Cleaned: landslide cause severe rainstorm kill italian alps
```

```
#Apply the text_cleaning function to all of the texts
df['clean_text'] = df['text'].apply(text_cleaning)
```

```
df.head(5)
```

	target	text	clean_text
0	Yes	Just happened a terrible car crash	happen terrible car crash
1	Yes	Our Deeds are the Reason of this #earthquake M...	deeds reason earthquake may allah forgive
2	Yes	Heard about #earthquake is different cities, s...	hear earthquake different cities stay safe eve...
3	Yes	there is a forest fire at spot pond, geese are...	forest fire spot pond geese flee across street...
4	Yes	Forest fire near La Ronge Sask. Canada	forest fire near ronge sask canada

Feature Engineering

As a final step to complete my dataset, new features were created. I used the cleaned text column to create [1] tweet length and [2] number of words feature and used the original text column to count the number of [3] hashtags and [4] mentions(@).

[1] Tweet Length

```
#tweet length feature
df['tweet_length'] = df['clean_text'].str.len()
```

```
df.head()
```

	target	text	clean_text	tweet_length
0	Yes	Just happened a terrible car crash	happen terrible car crash	25
1	Yes	Our Deeds are the Reason of this #earthquake M...	deeds reason earthquake may allah forgive	41
2	Yes	Heard about #earthquake is different cities, s...	hear earthquake different cities stay safe eve...	51
3	Yes	there is a forest fire at spot pond, geese are...	forest fire spot pond geese flee across street...	51
4	Yes	Forest fire near La Ronge Sask. Canada	forest fire near ronge sask canada	34

[2] Number of Words

```
#number of words feature

def word_count(text):
    words = text.split()
    return len(words)

df['num_words'] = df['clean_text'].apply(word_count)

df.head()
```

	target	text	clean_text	tweet_length	num_words
0	Yes	Just happened a terrible car crash	happen terrible car crash	25	4
1	Yes	Our Deeds are the Reason of this #earthquake M...	deeds reason earthquake may allah forgive	41	6
2	Yes	Heard about #earthquake is different cities, s...	hear earthquake different cities stay safe eve...	51	7
3	Yes	there is a forest fire at spot pond, geese are...	forest fire spot pond geese flee across street...	51	9
4	Yes	Forest fire near La Ronge Sask. Canada	forest fire near ronge sask canada	34	6

[3] Hashtag Count

```
#hashtag feature

def hashtag_count(text):
    words = text.split()
    hashtags = [word for word in words if word.startswith('#')]
    return len(hashtags)

df['hashtag_count'] = df['text'].apply(hashtag_count)

df.head()
```

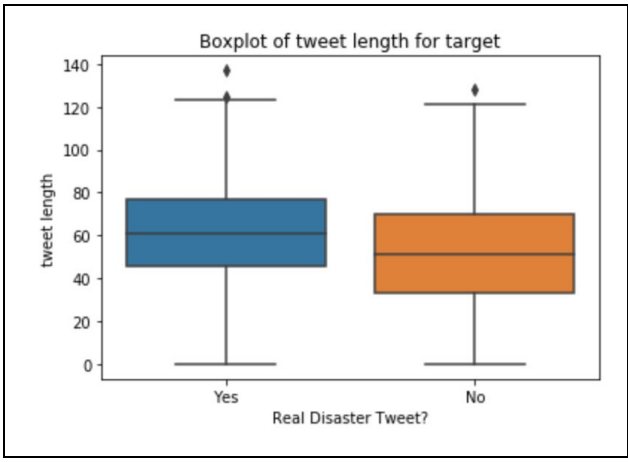
	target	text	clean_text	tweet_length	num_words	hashtag_count
0	Yes	Just happened a terrible car crash	happen terrible car crash	25	4	0
1	Yes	Our Deeds are the Reason of this #earthquake M...	deeds reason earthquake may allah forgive	41	6	1
2	Yes	Heard about #earthquake is different cities, s...	hear earthquake different cities stay safe eve...	51	7	1
3	Yes	there is a forest fire at spot pond, geese are...	forest fire spot pond geese flee across street...	51	9	0
4	Yes	Forest fire near La Ronge Sask. Canada	forest fire near ronge sask canada	34	6	0

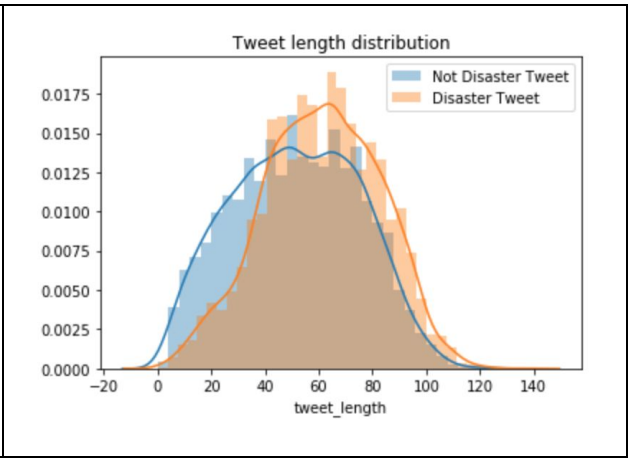
[4] Mention Count

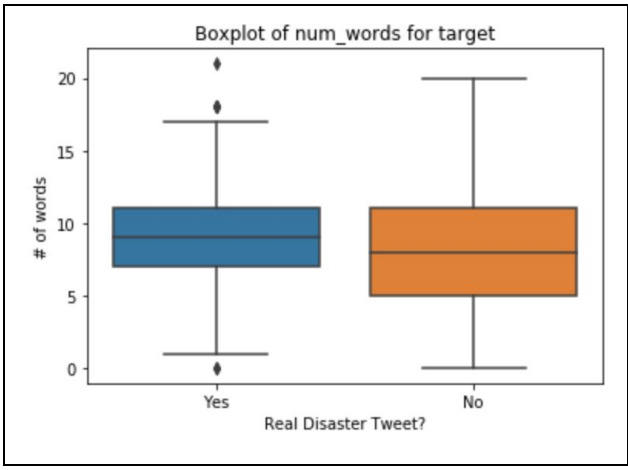
```
#mention feature

def mention_count(text):
    words = text.split()
    hashtags = [word for word in words if word.startswith('@')]
    return len(hashtags)

df['mention_count'] = df['text'].apply(mention_count)

df.head()
```

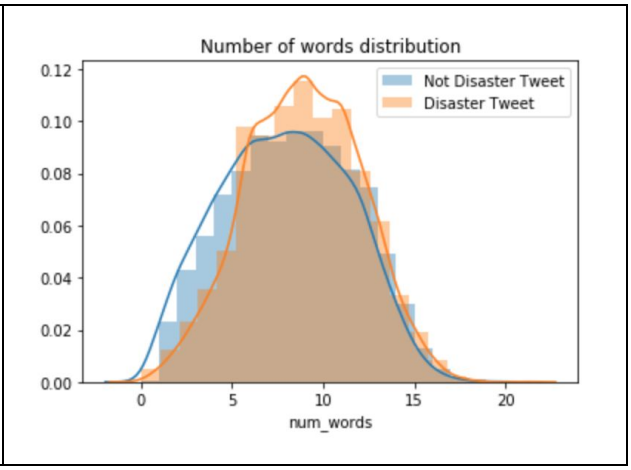
	target	text	clean_text	tweet_length	num_words	hashtag_count	mention_count
0	Yes	Just happened a terrible car crash	happen terrible car crash	25	4	0	0
1	Yes	Our Deeds are the Reason of this #earthquake M...	deeds reason earthquake may allah forgive	41	6	1	0
2	Yes	Heard about #earthquake is different cities, s...	hear earthquake different cities stay safe eve...	51	7	1	0
3	Yes	there is a forest fire at spot pond, geese are...	forest fire spot pond geese flee across street...	51	9	0	0
4	Yes	Forest fire near La Ronge Sask. Canada	forest fire near ronge sask canada	34	6	0	0

Exploratory Data Analysis





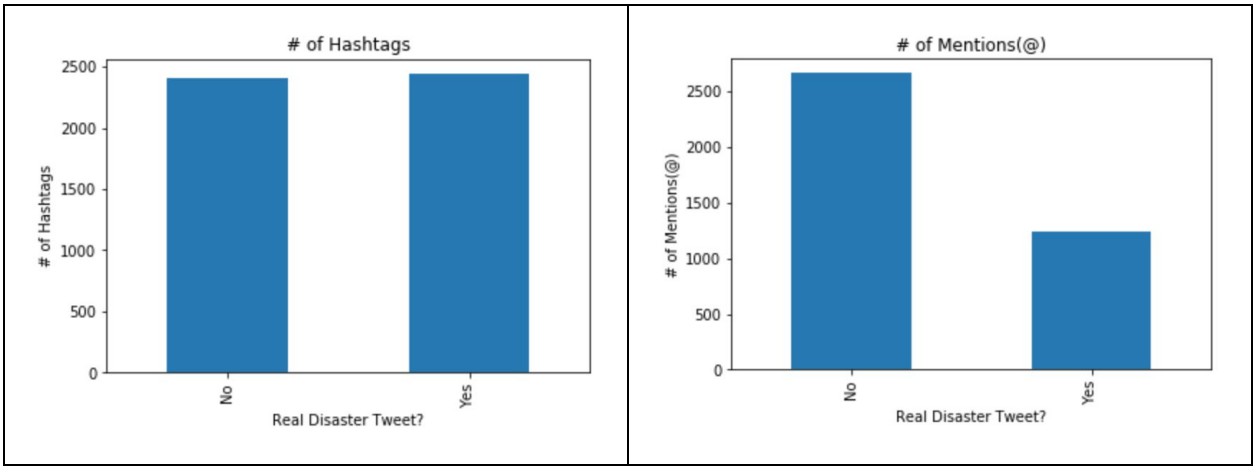




Disaster tweet lengths are slightly left skewed while non-disaster tweets are slightly right skewed. On average, real disaster tweets have longer tweet lengths and number of words. If we think of why that might be, it would make sense that a user's tweet is longer and has more words when they are trying to describe a situation that's going on, like a disaster. This would also make sense on why both distributions are slightly skewed. Tweets are short, 280 characters allowed, and tweets that are not trying to describe a situation, such as a disaster, would be short in length.

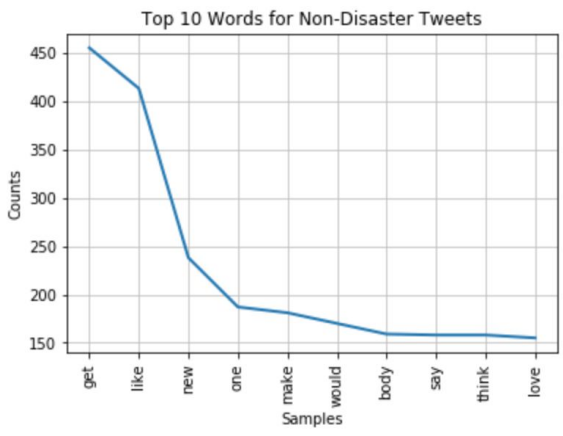
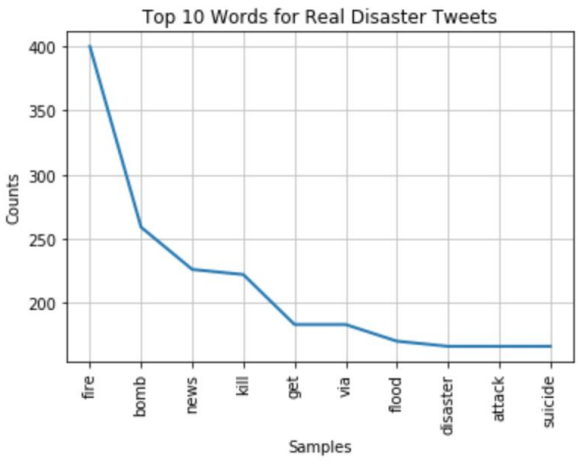
```
df[(df['tweet_length']<50) & (df['target']=='No')]
```

	target	text	clean_text	t
21	No	They'd probably still show more life than Arse...	probably still show life arsenal yesterday	
22	No	Hey! How are you?	hey	
23	No	What's up man?	man	
24	No	I love fruits	love fruit	
25	No	Summer is lovely	summer lovely	



The number of hashtags in each type of tweet is almost identical. I can see how both are helpful in both types of situations. # added before a word or phrase allows that word to be a searchable keyword in twitter. Users can better find posts by looking for certain keywords. The number of mentions are a lot higher for tweets that are not related to disasters. When users use @ before a username, it tags that person in that post.

For text analysis, I utilized Frequency Distribution and ngrams from the NLTK library to plot, find most common words, and convert texts to bigrams and trigrams.

Not Disaster Tweet	**Disaster Tweet**
	

```
[('get', 455),
 ('like', 413),
 ('new', 238),
 ('one', 187),
 ('make', 181),
 ('would', 170),
 ('body', 159),
 ('say', 158),
 ('think', 158),
 ('love', 155),
 ('see', 154),
 ('time', 151),
 ('bag', 151),
 ('come', 150),
 ('via', 142),
 ('know', 140),
 ('video', 135),
 ('want', 132),
 ('people', 132),
 ('burn', 130)]
```

```
[('fire', 400),
 ('bomb', 259),
 ('news', 226),
 ('kill', 222),
 ('get', 183),
 ('via', 183),
 ('flood', 170),
 ('disaster', 166),
 ('attack', 166),
 ('suicide', 166),
 ('crash', 152),
 ('people', 151),
 ('police', 151),
 ('california', 150),
 ('train', 147),
 ('hiroshima', 146),
 ('like', 146),
 ('home', 137),
 ('storm', 137),
 ('build', 134)]
```

The unigram model above displays the most commonly used words in each case of whether a tweet is related to a disaster or not. For disaster tweets, we see the top 2 words that relates to a disaster: “fire” and “bomb”. But seeing just these single words, like the word ‘california’ in the disaster words, it will be hard to categorize the tweet without previously knowing the target. This is where bigrams and trigrams come in handy because it lists pairs of words that occur together frequently.

Not Disaster Tweet	Disaster Tweet
[(('body', 'bag'), 99), (('cross', 'body'), 53), (('like', 'video'), 51), (('look', 'like'), 47), (('full', 'read'), 42), (('feel', 'like'), 38), (('loud', 'bang'), 31), (('burn', 'build'), 29), (('first', 'responders'), 29), (('content', 'policy'), 28), (('china', 'stock'), 27), (('stock', 'market'), 27), (('market', 'crash'), 27), (('emergency', 'service'), 26), (('reddit', 'quarantine'), 26), (('quarantine', 'offensive'), 26), (('offensive', 'content'), 26), (('take', 'quiz'), 23), (('break', 'news'), 23), (('read', 'ebay'), 22)]	[(('suicide', 'bomber'), 91), (('atomic', 'bomb'), 67), (('northern', 'california'), 57), (('suicide', 'bomb'), 52), (('oil', 'spill'), 50), (('california', 'wildfire'), 46), (('bomber', 'detonate'), 46), (('burn', 'build'), 44), (('detonate', 'bomb'), 43), (('pkk', 'suicide'), 43), (('wild', 'fire'), 42), (('year', 'old'), 41), (('old', 'pkk'), 41), (('mass', 'murder'), 40), (('severe', 'thunderstorm'), 40), (('home', 'raze'), 40), (('latest', 'home'), 39), (('raze', 'northern'), 39), (('heat', 'wave'), 36), (('bomb', 'turkey'), 36)]

Bigrams starts to add more context to the different types of tweets. For disaster tweets, we can now see that the unigram word ‘california’ is frequently paired with ‘wildfire’. This makes sense as wildfire disasters have been prevalent in California the past few years. “Raze”, another word for destroy, occurs with home and northern, which when I looked up on Google had a lot of

references to wildfires in Northern California that razed houses. ‘PKK’ refers to the Kurdistan Workers Party, who are broadly labeled as terrorists, is frequently paired with ‘suicide’. In the tweets that are not a disaster, the word “body” and “bag” are common together. This seems odd to me because when I see body and bag together, I assume a bodybag used to place the deceased.

Not Disaster Tweet	Disaster Tweet
[(('cross', 'body', 'bag'), 33), (('china', 'stock', 'market'), 27), (('stock', 'market', 'crash'), 27), (('reddit', 'quarantine', 'offensive'), 26), (('quarantine', 'offensive', 'content'), 25), (('full', 'read', 'ebay'), 22), (('break', 'news', 'unconfirmed'), 22), (('news', 'unconfirmed', 'hear'), 22), (('unconfirmed', 'hear', 'loud'), 22), (('hear', 'loud', 'bang'), 22), (('loud', 'bang', 'nearby'), 22), (('bang', 'nearby', 'appear'), 22), (('nearby', 'appear', 'blast'), 22), (('appear', 'blast', 'wind'), 22), (('blast', 'wind', 'neighbour'), 22), (('wind', 'neighbour', 'ass'), 22), (('reddit', 'new', 'content'), 22), (('new', 'content', 'policy'), 22), (('content', 'policy', 'effect'), 21), (('policy', 'effect', 'many'), 21)]	[(('suicide', 'bomber', 'detonate'), 46), (('pkk', 'suicide', 'bomber'), 42), (('bomber', 'detonate', 'bomb'), 42), (('northern', 'california', 'wildfire'), 41), (('old', 'pkk', 'suicide'), 41), (('latest', 'home', 'raze'), 39), (('home', 'raze', 'northern'), 39), (('raze', 'northern', 'california'), 38), (('severe', 'thunderstorm', 'warn'), 36), (('families', 'affect', 'fatal'), 34), (('affect', 'fatal', 'outbreak'), 34), (('watch', 'airport', 'get'), 34), (('airport', 'get', 'swallow'), 34), (('get', 'swallow', 'sandstorm'), 34), (('swallow', 'sandstorm', 'minute'), 34), (('detonate', 'bomb', 'turkey'), 34), (('bomb', 'turkey', 'army'), 34), (('turkey', 'army', 'trench'), 34), (('families', 'sue', 'legionnaires'), 33), (('wreckage', 'conclusively', 'confirm'), 33)]

Looking at a trigram model, we now see that “body” and “bag” in the not disaster tweets had a high occurrence with “cross”, which could refer to a popular handbag; cross body bags. ‘California’ and ‘wildfire’ from the bigram model now includes ‘northern’. This gives a better idea to where in California the wildfires may be taking place. One thing noticeable is the count of occurrences. As we increase the models from unigram to bigram to trigram, we see that occurrences of words decrease. I will use all 3 types in my models to see which gives the best results.

Further Preprocessing

There are a few steps that need to be done before feeding the models with the cleaned data:

1. I separated feature and target values and then encoded the target values to 1 (yes) and 0 (no).

```
features = df[['clean_text','tweet_length', 'num_words','hashtag_count', 'mention_count']]
target = df['target']

#Convert target to 1(yes) and 0(no)
Encoder = LabelEncoder()
target = Encoder.fit_transform(target)
```

2. Data was then split 70/30 between training and test, respectively.

```
#Split into train and test

X_train, X_test, y_train, y_test = train_test_split(features, target, test_size=0.3, random_state=34)
```

3. The cleaned text needs to be transformed into integers or floats before being used by each model. This will be done by using CountVectorizer. Depending on the type of model being used, the other features that are not text will have to be rescaled.

```
#Use ColumnTransformer to separate the normalizer for numerical column from text vectors

counts = ['tweet_length', 'num_words','hashtag_count', 'mention_count']
texts = 'clean_text'

preprocessor = ColumnTransformer(
    transformers = [('cv', CountVectorizer(), texts)],
    remainder = MinMaxScaler() )

#Transformer for Decision Tree and Random Forest since numerical values do not need to be normalized.

tree_preprocessor = ColumnTransformer(
    transformers = [('cv', CountVectorizer(), texts)],
    remainder = 'passthrough' )
```

Model Performance Testing

Various machine learning models were used for classification: Logistic Regression, SVC, K-Nearest Neighbors, Decision Tree Classifier, Random Forest Classifier, AdaBoost and Gradient Boost Classifier. To test the performance of each classifier, a baseline model was created using DummyClassifier.

```
dummy = make_pipeline(preprocessor, DummyClassifier(random_state=34))
dummy_acc = dummy.fit(X_train, y_train).score(X_test,y_test)
dummy_f1 = f1_score(y_test, dummy.fit(X_train,y_train).predict(X_test))

print ('Baseline Model Accuracy for CountVectorizer: %.3f' % dummy_acc)
print ('Baseline Model F1-score for CountVectorizer: %.3f' % dummy_f1)
print(confusion_matrix(y_test, dummy.fit(X_train,y_train).predict(X_test)))
```

```
Baseline Model Accuracy for CountVectorizer: 0.505
Baseline Model F1-score for CountVectorizer: 0.416
[[1071  777]
 [ 836  574]]
```

The goal is for the models to perform better than the baseline model, measuring against f1-score, a measure of the weighted average between precision and recall. For this problem, it is important for us to focus on False Negatives and False Positives. We want to make sure that we can decrease the amount of disaster tweets that get classified as a non-disaster.

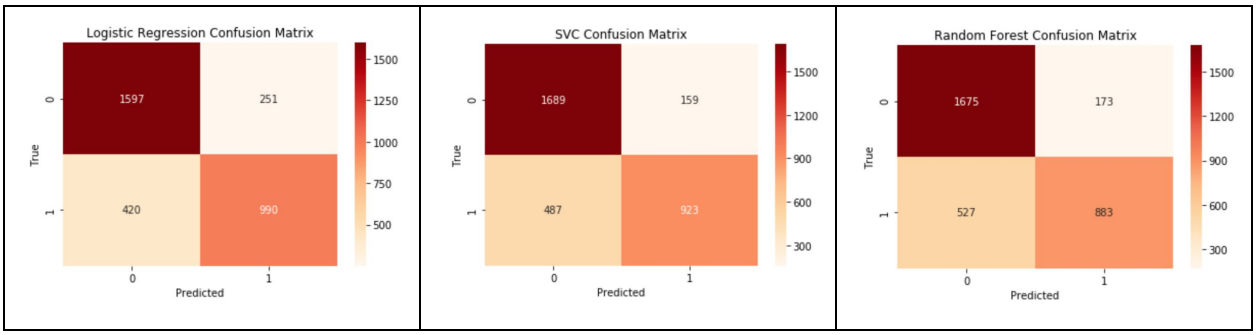
Each model will be used in a pipeline with the transformers.

```
lr_pipe = make_pipeline(preprocessor, LogisticRegression(random_state=34))
lr_pipe.fit(X_train, y_train)
y_pred = lr_pipe.predict(X_test)
```

```
#Sorted F1 score

model_performance['f1_score'].sort_values(ascending = False)
```

```
CountVectorizer  LogReg           0.746888
                 SVC              0.740770
                 RandomForest     0.716139
                 DecisionTree     0.685457
                 GradientBoost    0.638070
                 KNN              0.627281
                 AdaBoost         0.624733
```

The top performing models were Logistic Regression, SVC, and Random Forest Classifier. Logistic Regression performed the best with a f1-score of 0.75. We can see that the disaster tweets that were categorized as non-disaster is the lowest out of the 3 at 420 wrongly classified.



CountVectorizer vs TfidfVectorizer

CountVectorizer was used to do a preliminary check on all the models. As CountVectorizer counts the word frequencies, TfidfVectorizer, Term Frequency Inverse Document Frequency Vectorizer, considers the overall weight of a word throughout the document. This penalizes words that appear too frequently.

There was a small increase in performance for all the models when using TfidfVectorizer. This will be the vectorizer that I use for model tuning.

Machine Learning Model Tuning

RandomizedSearchCV will be used to find optimal parameters for TfidfVectorizer and for each model.

TfidfVectorizer Parameters

ngram_range: Default ngram_range for TfidfVectorizer is (1,1), which means that it uses unigram only. I tested out various ngrams such as unigrams only, unigram and bigram mix (1,2), and unigram and trigram mix (1,3).

min_df: Used to remove terms that appear infrequently. Using int value will ignore terms that appear in less than X amount of documents. Using a float value will ignore terms that appear in less than X% of the documents.

max_df: Used for removing terms that appear too frequently. Using int value will igmore terms that appear in more than X amount of documents. Using a float value will ignore terms that appear in more than X% of the documents.

Logistic Regression Tuning

penalty: This helps reduce the model complexity and prevent overfitting by placing constraints on large coefficients, essentially shrinking it. By default, the penalty is set to 'L2', ridge regression. Ridge regression puts constraints by using the sum of squares of the coefficients. Another available option is 'L1', Lasso Regression, that puts constraints by using the sum of the absolute values of the coefficients.

C: This is the inverse of regularization strength. Smaller C value will increase regularization strength, which can lead to underfitting the data. Bigger C values will decrease the regularization strength, allowing the model to increase it's complexity and overfit the data.

max_iter: Maximum number of iterations taken for the solvers to converge.

```
%%time
lr_param = {
    'columntransformer__tfidf__ngram_range': [(1,1),(1,2),(1,3)],
    'columntransformer__tfidf__min_df':[1, 5, 10],
    'columntransformer__tfidf__max_df':[0.5, 0.75, 0.95],
    'logisticregression__penalty': ['l1', 'l2', None],
    'logisticregression__C': [1, 25, 50, 100],
    'logisticregression__max_iter': [10, 100, 500, 1000],
}
random = RandomizedSearchCV(estimator = lr_pipe_tfidf,
                            param_distributions = lr_param,
                            cv=5,
                            scoring = 'f1',
                            random_state=34)
```

Logistic Regression Performance

```
Best Score:  0.7525072251167785

Best Params:  {'logisticregression__penalty': 'l2', 'logisticregression__max_iter': 10, 'logisticregression__C': 50,
'columntransformer__tfidf__ngram_range': (1, 1), 'columntransformer__tfidf__min_df': 5, 'columntransformer__tfidf__ma
x_df': 0.75}
CPU times: user 24.7 s, sys: 922 ms, total: 25.6 s
Wall time: 29 s
```

For Logistic Regression, the TFIDF Parameters that worked the best were unigrams only, ignoring words that appear in less than 5 documents and ignoring words that appear in more that 75% of the document. This model performed pretty quick, as well.

SVC Tuning

C: Same as the C parameter of Logistic Regression

kernel: This selects the type of hyperplane used to separate the data. 'linear' uses a linear hyperplane. 'rbf' stands for Radial Basis Function and uses a non-linear hyperplane.

gamma: This helps define the 'spread' of the kernel. When gamma is high, the values try to exactly fit the the data set, this can lead to overfitting.

```
%%time
svc_param = {
    'columntransformer__tfidf__ngram_range': [(1,1),(1,2),(1,3)],
    'columntransformer__tfidf__min_df':[1, 5, 10],
    'columntransformer__tfidf__max_df':[0.5, 0.75, 0.95],
    'svc__C': [1, 25, 50, 100],
    'svc__kernel': ['linear', 'rbf'],
    'svc__gamma': [0.01, 0.1, 1, 10, 100]
}
random = RandomizedSearchCV(estimator = svc_pipe_tfidf,
                            param_distributions = svc_param,
                            cv=5,
                            scoring = 'f1',
                            random_state=34)
```

SVC Performance

```
Best Score:  0.7505131014008629

Best Params:  {'svc__kernel': 'rbf', 'svc__gamma': 0.1, 'svc__C': 25, 'columntransformer__tfidf__ngram_range': (1,
3), 'columntransformer__tfidf__min_df': 1, 'columntransformer__tfidf__max_df': 0.75}
CPU times: user 6min 50s, sys: 7.2 s, total: 6min 58s
Wall time: 8min 49s
```

For SVC, the TFIDF Parameters that worked the best were a unigram/trigram mix, not igoring any words that show up less than X amount of times, and ignoring words that appear in more that 75% of the document. This model took longer than Logistic Regression to process.

Random Forest Classifier Tuning

n_estimators: This is the number of trees in the forest.

max_features: Maximum number of features considered for splitting a node.

max_depth: Maximum number of levels in each tree

```
%%time
rf_param = {
    'columntransformer__tfidf__ngram_range': [(1,1),(1,2),(1,3)],
    'columntransformer__tfidf__min_df':[1, 5, 10],
    'columntransformer__tfidf__max_df':[0.5, 0.75, 0.95],
    'randomforestclassifier__n_estimators': [50, 100, 250, 500],
    'randomforestclassifier__max_features': ['sqrt', 'log2', None],
    'randomforestclassifier__max_depth': [50, 100, 250, 500, None],
}
random = RandomizedSearchCV(estimator = rf_pipe_tfidf,
                            param_distributions = rf_param,
                            cv=5,
                            scoring = 'f1',
                            random_state=34)
```

Random Forest Classifier Performance

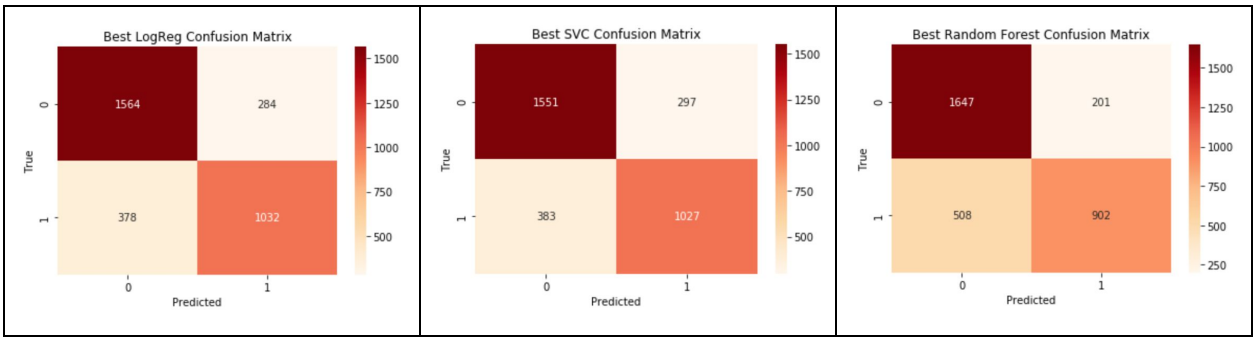
```
Best Score:  0.7221987051533116

Best Params:  {'randomforestclassifier__n_estimators': 50, 'randomforestclassifier__max_features': 'sqrt', 'randomfor
estclassifier__max_depth': 250, 'columntransformer__tfidf__ngram_range': (1, 3), 'columntransformer__tfidf__min_df':
5, 'columntransformer__tfidf__max_df': 0.5}
CPU times: user 28min 20s, sys: 4.3 s, total: 28min 24s
Wall time: 53min 31s
```

For Random Forest Classifier, the TFIDF Parameters that worked the best were a unigram/trigram mix, ignoring any words that show up less than 5 times in a document, and ignoring words that appear in more that 50% of the document. This model took longest to run.

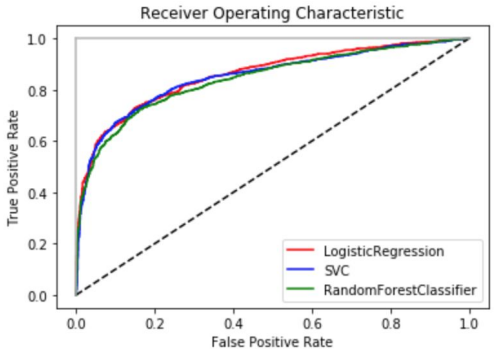
Model Selection

When looking at the F-1 scores for the 3 models, Logistic Regression came out to perform the best. The processing time was fast and the amount of False Negatives are the lowest of all the models. This is important because we want to make sure that actual tweets that are disasters get categorized correctly and do not go unnoticed. Logistic Regression also had the highest accuracy score.

Best F1 Score		Best Accuracy Score	
LogReg	0.757153	LogReg	0.796808
SVC	0.751280	SVC	0.791283
RandomForest	0.717867	RandomForest	0.782382



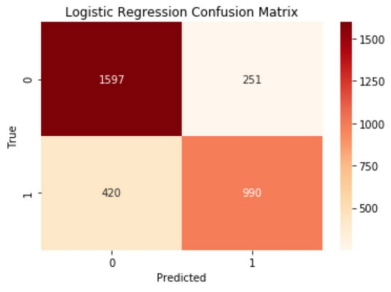
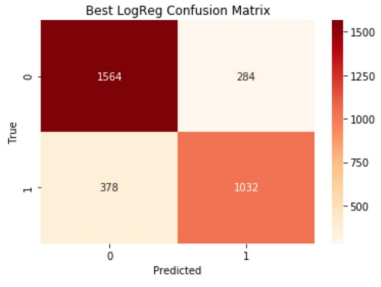
The area under the ROC curve was measured for each model. Logistic Regression slightly outperforms SVC in this metric.




```
ROC AUC Score - LogReg:  0.8587767876331706
ROC AUC Score - SVC:  0.8507548893187191
ROC AUC Score - RandomForest:  0.8417010147063337
```

When it comes down to it, the Logistic Regression model performed the best in terms of f1-score, low false negatives, and time.

Conclusion and Future Work

Original	Tuned
	

Logistic Regression model performed the best when predicting whether a tweet refers to an actual disaster or not. Along with TFIDF Vectorizer providing the best results when applying

weights to term importance, this model was able to produce an accuracy of about 80% with an f1-score of 0.76. The confusion matrix shows an improvement in false negatives, this is ideal because we always want to make sure that actual disaster tweets are getting labeled as such.

Being that this project was time-sensitive, I would've liked to use Grid Search for a more exhaustive search means to tune the parameters. More time could've been taken by cleaning the text. There were some words that had multiple letters in it, for example 'aaaand' and 'and' are the same word but lemmatizing could not solve this. Acronyms could've been handled better and could've provided more insight. For example, 'LA' was cleaned to display 'la' but I would've rather seen it as 'los angeles'. I would've liked to use more features like time and location of tweet to see how that can affect the model.