Capstone 2 Final Report

James Olmstead

The Problem

For my second capstone project, I'd like to utilize a different field of machine learning from my first. To this end, I'd like to utilize the natural language toolkit (nltk) library in python to create a fake news classifier. The problem of fake news is quite relevant with the proliferation of social media and with the onslaught of news from a wide range of reputable or not reputable sources. Having a fake news classifier can help social media platforms or other entities flag potentially questionable news stories and either prevent its distribution or warn viewers of its falsity.

The Data

The data comes from the Kaggle competition: https://www.kaggle.com/c/fakenewskdd2020. There are 3 raw data sets that come from Kaggle that I'll be using, with descriptions below directly from the website:

train.csv: A full training dataset with the following attributes:

- text: text of the article
- label: a label that marks the article as potentially unreliable
    - 1: fake
    - 0: true

test.csv: A test dataset with the following attributes:

- id: the id of the article text
- text: text of the article

sample_submission.csv: A sample submission with the following attributes.

- id: the id of the article text from the test data
- label: a label that marks the article as potentially unreliable
    - 1: fake
    - 0: true

Note that the *test.csv* file doesn't have labels. Since this was a Kaggle competition, I'll measure my performance of my label predictions of the test set by submitting it to the website in the format demonstrated in *sample_submission.csv*. Then it will grade the percentage of labels I predicted correctly. I can also compare my performance to the leaderboard to get a gage on how well I do in predicting the labels.

Data Cleaning

Since this data was pre-made for the competition, the amount of cleaning I needed to do was minimal. The raw test.csv file initially had all of the articles separated into different cells every time there was a comma, causing some headaches with how it was being read into the notebook. So, the first thing I did was convert it to a .txt file. This allowed me to read it neatly without needing to worry about all the commas. The only other real snag was I found that one of the labels in the train data were labeled as the letter "l" instead of the number "1", so I changed that. Finally, I changed the labels from "0" to "Real" and "1" to "Fake" to give the labels a more intuitive meaning.

Data Pre-processing

The next step with the data was to break the texts down into simpler components that may otherwise clutter the algorithm. This is called pre-processing, and for this project can be broken down into 4 parts:

1. Convert everything to lower-case. This way the algorithm won't distinguish between "One" and "one", for example.
2. Remove special characters like question marks, commas, and periods. This way the algorithm won't distinguish between "end" and "end!", for example.
3. Remove stop words, which are common words that don't add anything really meaningful such as "the" and "of". This helps focus the algorithm on the essential words

that make up the meat of the article. The stop words I removed were from the nltk.corpus English library.

4. Apply a stemmer to all the words. This is a bit technical, but it essentially boils down a word to its root. An example would be “going” and “goes” being changed to “go”. There are a few ways to do this stemming, but for this project I used a Porter Stemmer.

Once all of these steps are finished, the given article is completely pre-processed. Below is an example of a sentence from the data before and after this pre-processing:

- `Much like a certain Amazon goddess with a lasso, there are no heights that director Patty Jenkins can’t scale.`
- `much like certain amazon goddess lasso height director patti jenkin scale`

Data Vectorization

The final step before we can create a model is to vectorize the text data. Essentially this is the process by which we convert text to numbers for the model to read in. I experimented with 2 different ways to do this, a count vectorizer and a Tf-Idf vectorizer. The count vectorizer only looks at the frequency of a word a given document, while the Tf-Idf vectorizer looks at the frequency of a word in the whole document as well.

Initial Exploratory Analysis

The first thing I wanted to look at was to see the training and test sizes and what proportion of the training set was fake and real. This can be seen in figure 1 on the left. These proportions seem pretty standard for a train/test split, with the test set being about

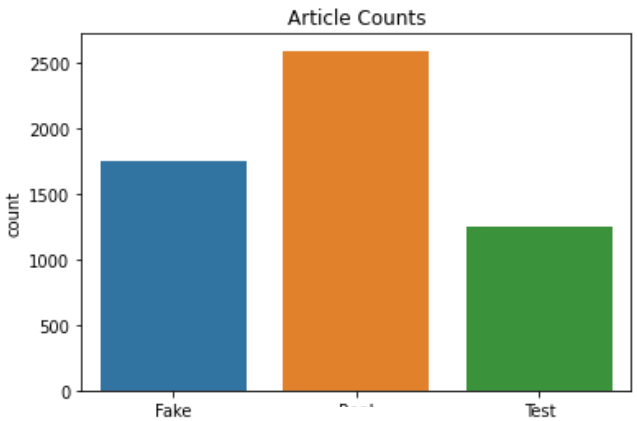


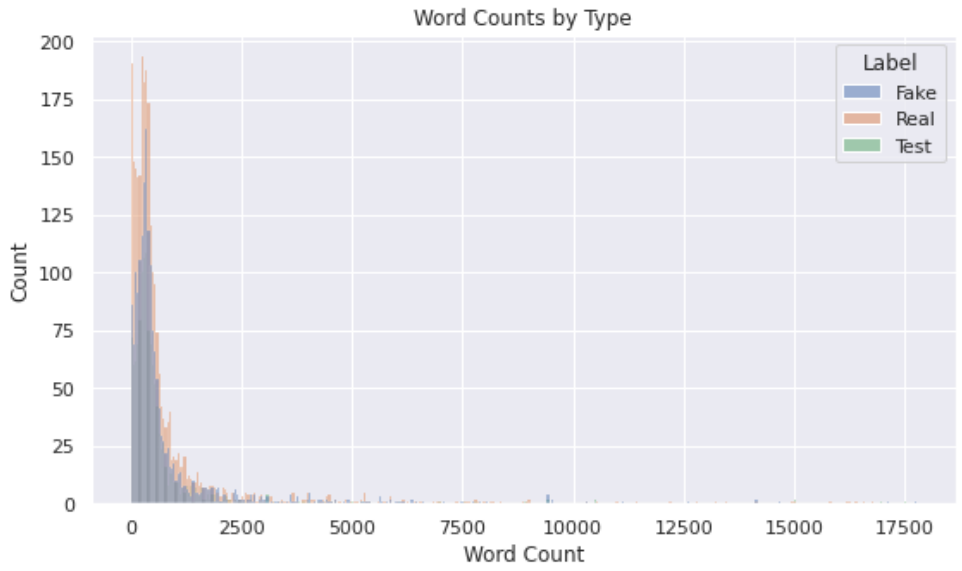
Figure 1

22% of the total data. While I don't have access

to the true labels for the test data, I imagine

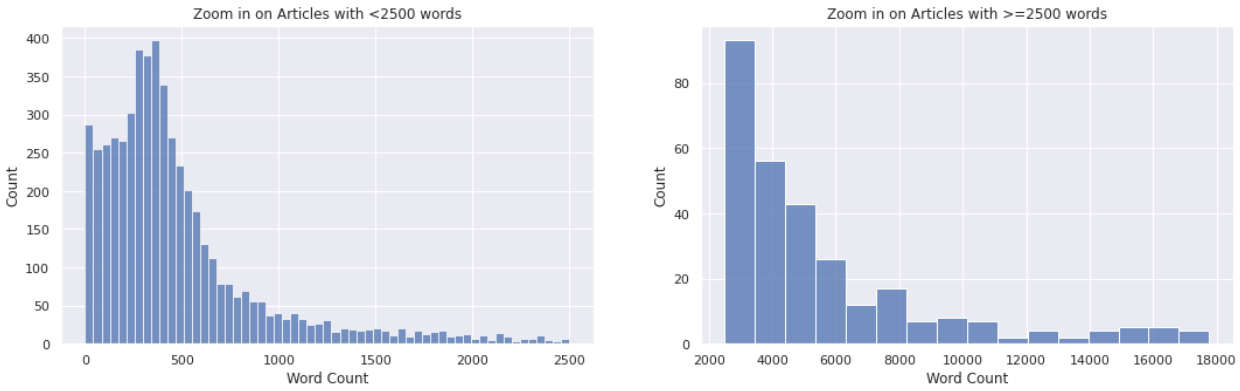
they will be similarly proportioned to the train set (about 60-40 real-fake).

Next, I wanted to examine the word counts of the articles to see the overall distribution of word counts, and to see if there were any meaningful differences between the training articles that were fake news, real news, and in the test set. This can be seen in figure 2.



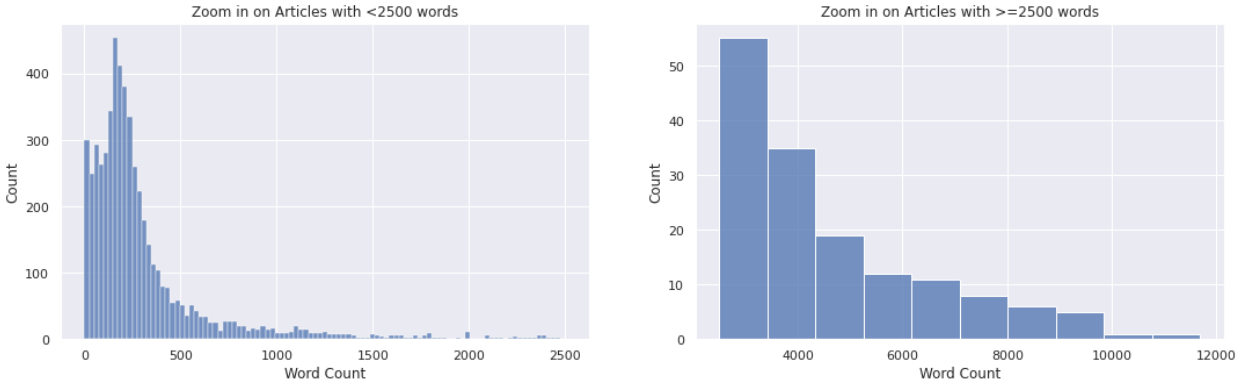
The only difference in the distributions between the 3 categories of articles seems to be just the number of articles in each category. Therefore, it seems safe to put them all together for purposes of examining word counts. I also found it useful to separate the main group of word counts (<2500 words) from the skinny tail (>= 2500) to get a better view of the distribution. Note that the 2500 number is simply a subjective choice by me. See figure 3.

Figure 3



Finally, I decided to look at the word counts before and after the pre-processing to see how much of each article was shaved off during the process. In examining the distribution of word counts, I found it useful to zoom in again on the main group (<2500 words) and the tail (>=2500 words). See figure 4.

Figure 4



The word counts definitely are distributed closer to 0, which is what I expect since we are cutting stop words and such in general. I also wanted to see how each article's word count changed. See figure 5.

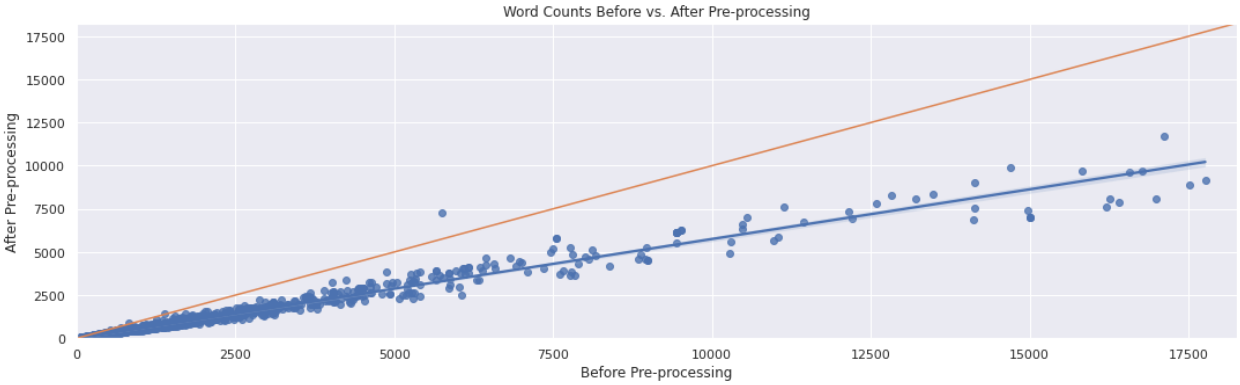


Figure 5

In the chart above, each point is an article in either the training or test data. Its position on each axis represents its word count before (x-axis) and after (y-axis) pre-processing. The green line is where x=y, or where articles would be if pre-processing wouldn't change the word count, while the blue line is the regression line. This plot shows mostly what is expected, namely that the pre-processing cuts the number of words. However there seems to be one glaring exception, and possibly more. After some further investigation, I found that there were about 11 articles where this was the case. And after looking at the text of each of those articles, there are 3 reasons why the pre-processing increased the word count.

1. A long website link that the pre-processor breaks down into multiple words
2. Poll data with connected numbers (like dates) that the pre-processor reads as separate numbers.
3. An article in Russian. Not quite sure exactly why but it is likely because we used English stop words.

The Model

I choose to use the Multinomial Naive Bayes model as it seemed to be the most commonly used model from my research on other fake news classifiers. In the future I might try other models to see if they perform better, but I only use the multinomial NB for purposes of this project. I also use a cross-validation grid search to tune a few basic parameters to improve the model performance.

The last part of creating my model is to see if we can reduce the dimensionality of the problem. In other words, I'd like to reduce the number of words in the model that aren't very useful in predicting the fake/real label. This is similar to removing the stop words in the pre-processing, but instead I remove words that aren't predictive. First, I needed to figure out which words to cut without ruining the model. The metric I use to see which words to cut is the chi-squared statistic. This measures how evenly distributed real and fake news articles are for a given word. For example, if 60% of the total articles are

real and 40% are fake, but 90% of the articles with a given word are real and 10% are fake, it will have a high chi-squared statistic.

Once that I had chi-squared stats and p-values for each word, I removed sets of words above a certain p-value and see what happened to the model. I used a set of metrics (accuracy, precision, recall, and f-score) to see what happened to the train and test sets as I removed sets of words. Those results are below in figures 6 and 7.

Accuracy scores behave mostly how we would expect. Generally fewer words mean a less accurate model, but less distance between train and test score. However, the test score actually starts to get slightly worse (at around p=.6) as we include more words in the model. Likely this is a result of overfitting.

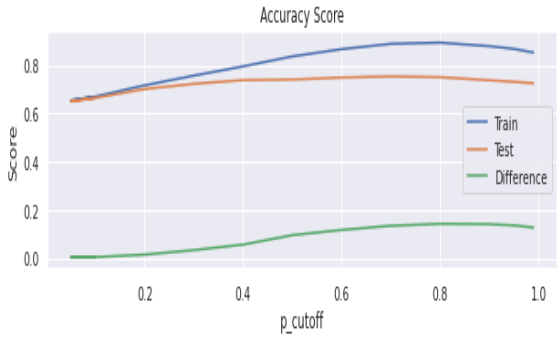


Figure 6

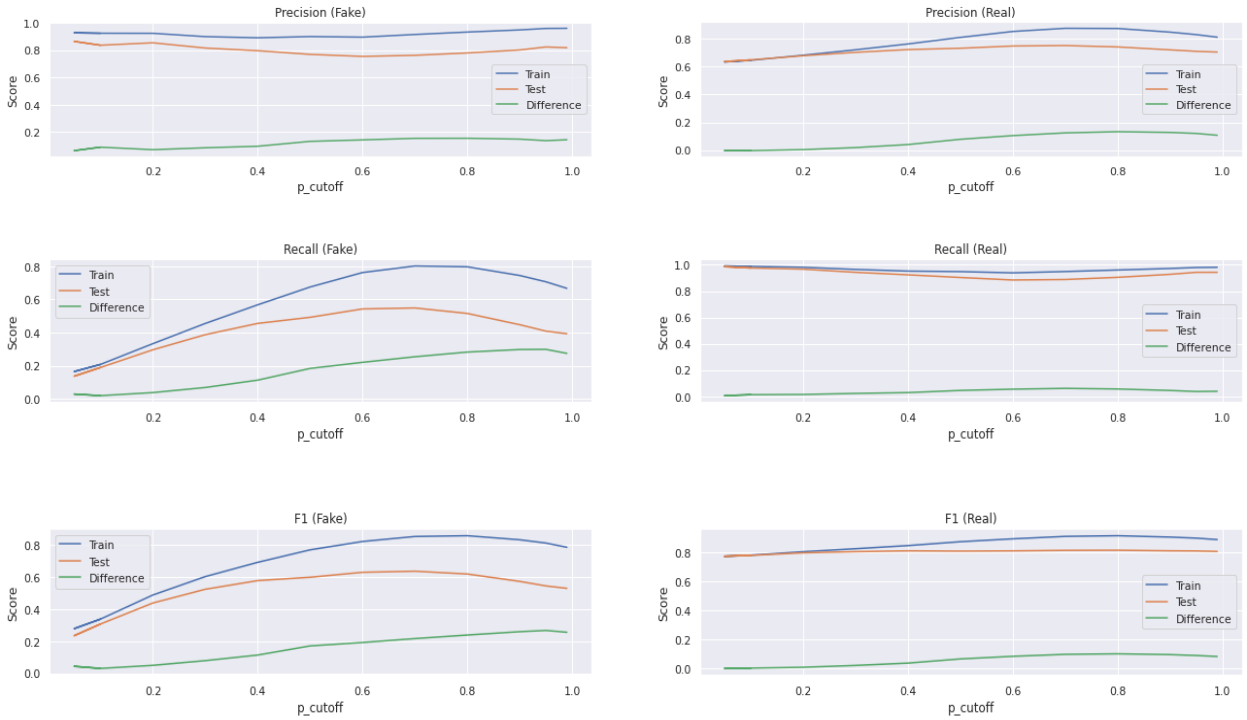


Figure 7

Using figures 6 and 7, I choose a p-value to align with 3 competing goals:

1. Minimize p-value (remove as many words as possible).

2. Maximize train and test scores (best performing model).

3. Minimize distance between train and test scores (model performance consistent when introducing new data).

By examining the above plots with our 3 goals in mind, I ended up going with .7 for my p cutoff value. It is the best f1 for fake news (which was the bigger problem between the 2 classes.) I chose to prioritize f1 over just precision or recall since both of them seem important for this type of problem in my opinion. (Incorrectly labeling a fake news article as real seems about as problematic as labeling a real news article as fake.) Coincidentally, this .7 is the same as what I'd choose by just looking at accuracy. My guess this is because the disparity in frequencies between real/fake classes isn't that large (only 60-40 real-fake). By removing words with p<.7, I only had 6601 of the original 39343 words, which is only about 17% of the original vocabulary! This should greatly reduce the likelihood of overfitting while not sacrificing too much in model performance.

Results and Analysis

The main way I wanted to analyze my final model was to see which words had the biggest impact on predicting the veracity of the articles. I accomplished this in a few different ways. The first way was to simply use my chi-squared stats from the feature selection. The second metric was the empirical log-probability. These number are all negative, and numbers closer to 0 signify a higher probability that, given a certain class, a given word is present in that class. For example, if an article is real, the word 'show' is relatively very likely compared to other words to appear in that article with a log-prob of -5.474328. The final way was to look at confusion matrices for articles with a given word to

see how closely that word matched up with either real or fake articles, and how well it predicted if it was real or fake. Below is an example of this for the train and test sets.

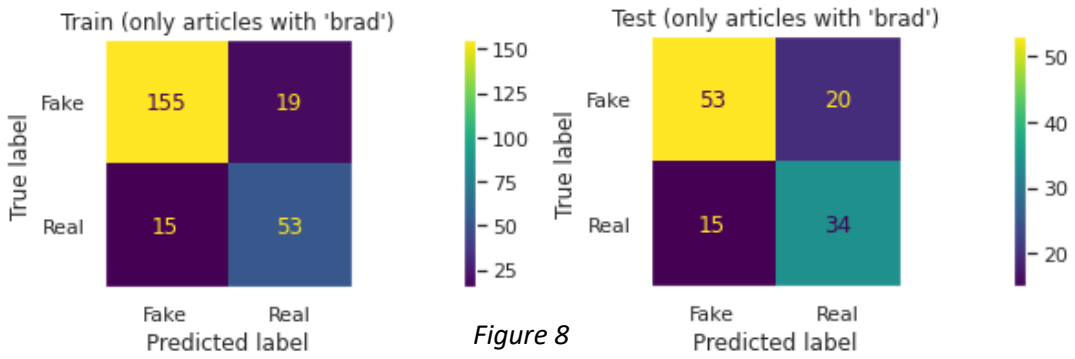


Figure 8

As you can see in Figure 8, there is a hugely disproportionate number of fake news articles with the word 'brad' (referring to Brad Pitt). Figure 8 also shows a moderate skew towards predicting fake articles as real (type 2 error) in the test set.

Figure 9 below shows the 15 words with the highest chi-squared value and the other stats I elected to use. Words with high chi-squared are more skewed, while the higher log-prob values indicate the direction of that skewness.

Rank	Word	chi2	Log_Prob_Fake	Log_Prob_Real
1	brad	35.20526	-5.594774	-8.018549
2	pitt	31.76604	-5.687666	-8.125464
3	source	20.94737	-5.459801	-6.64369
4	joli	18.9824	-6.016032	-7.906682
5	angelina	17.17027	-6.259558	-8.487052
6	aniston	14.86573	-6.279318	-8.189229
7	caitlyn	12.99058	-6.625143	-9.130981
8	insdie	12.44522	-6.053616	-7.349737
9	jen	12.40581	-6.323265	-7.866155
10	kany	11.40586	-6.190414	-7.531865
11	jenner	11.36223	-5.840508	-6.829059
12	gwen	11.04115	-6.624384	-8.594044
13	justin	10.73053	-5.943334	-6.950053
14	season	10.42455	-6.898985	-5.537182
15	hollywood	9.83713	-6.961444	-9.947012

Figure 9

Most of the words with high chi2 values seem to correspond more closely to fake news (with the exception of 'season'). This may indicate that most of the most impactful words appear in fake articles. Additionally, words in fake articles don't appear at the same rate in real articles as words in real articles appearing in fake ones. For example, a fake article may have the words "say", "go" and "brad", while a typical real article may only have the words "say" and "go". This makes it tricky to identify fake news articles without these flagging words.

Conclusion

Overall, I'm pleased with how this Capstone project ended up. Of course, the model didn't end up placing that well compared to the other Kaggle submissions, but it was still a valuable learning experience. If I were to work more on improving the model in the future, I'd definitely try other algorithms besides Multinomial Naive Bayes like a random forest, logistic regression, or SVM. I'd also try to implement some sort of named-entity recognition to capture all of the celebrity names better.