Capstone 2 Milestone Report 1

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

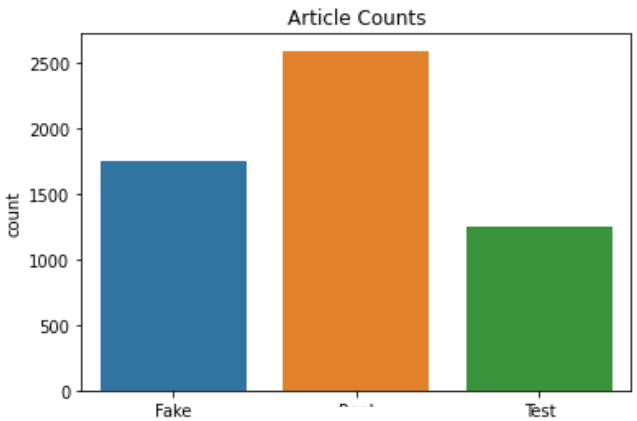


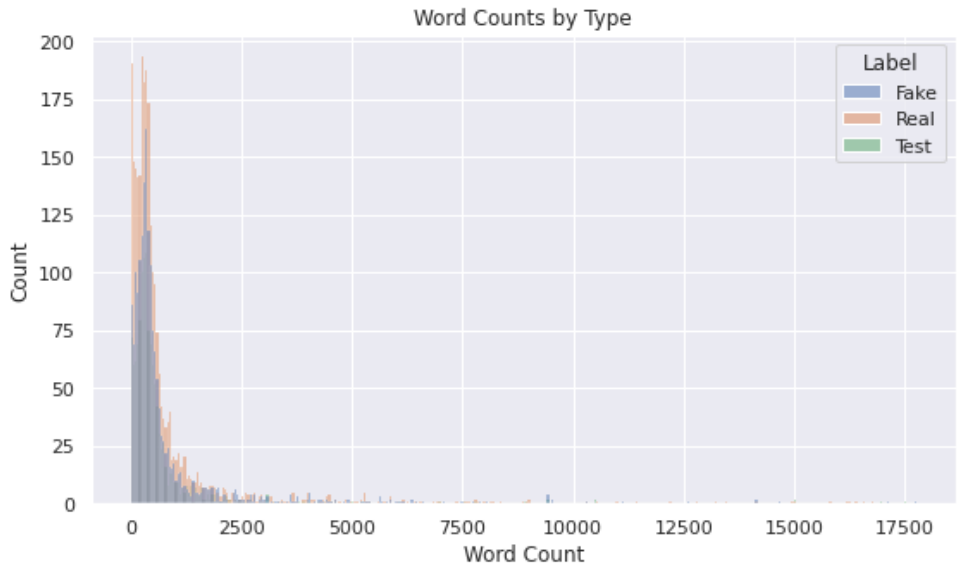
Figure 1

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

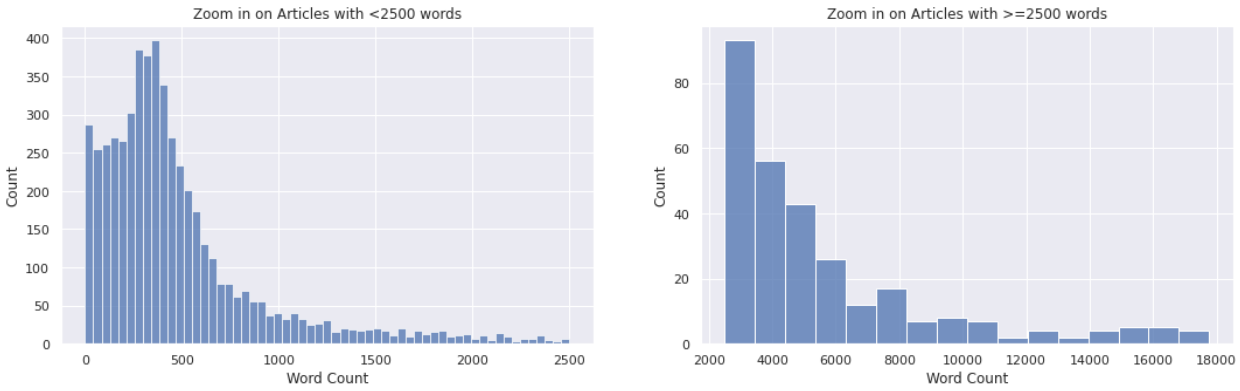
to the true labels for the test data, I imagine

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.

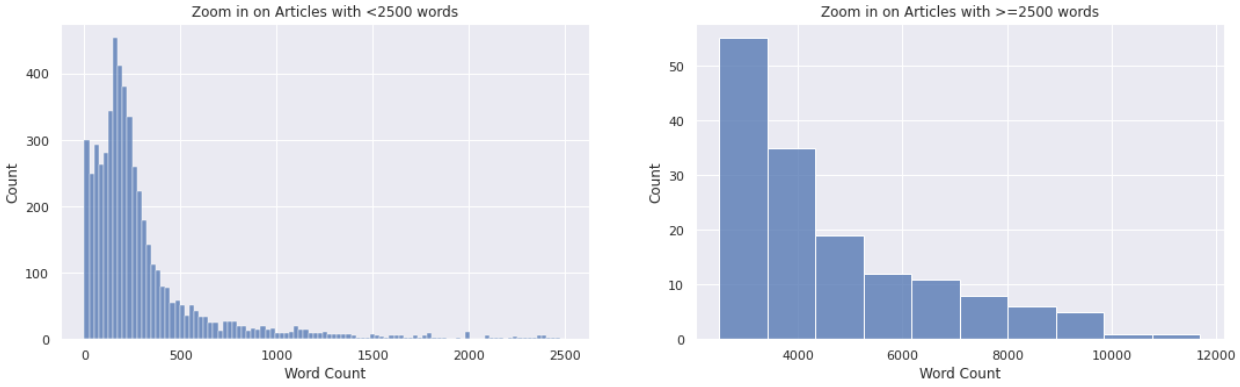


The only difference in the distributions between the 3 categories of articles seems to be just the number of articles in each category. Therefore, it seems safe to put them all together for purposes of examining word counts. I also found it useful to separate the main group of word counts (<2500 words) 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.



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

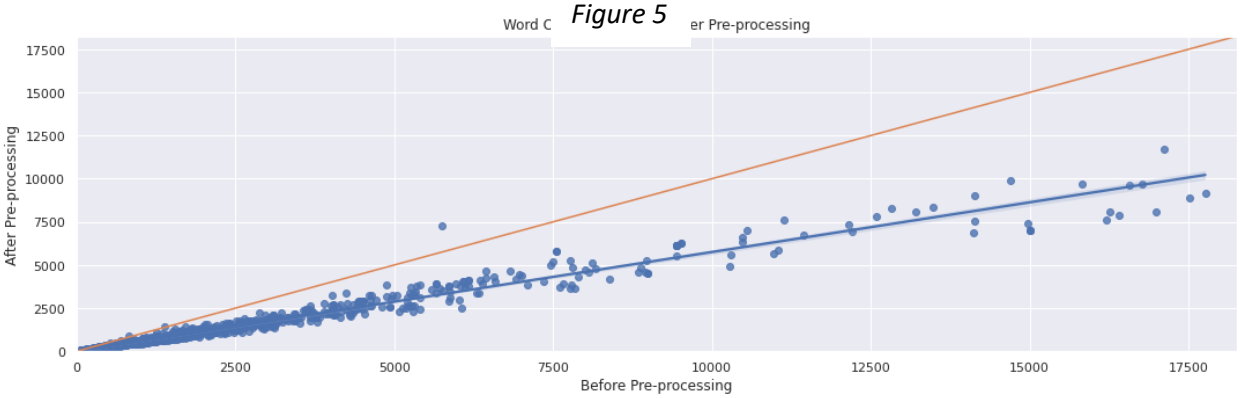


Figure 3

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.

Conclusion

Overall, the data was pretty clean to begin with so it didn't require too much work to clean. I was also able to apply standard text pre-processing so make the article text more digestible for the model. I look forward to see how the model performs with this data and see how it stacks up to the Kaggle leaderboard.