Fake News Detection using Natural Language Processing

Mohamed Ayman[†]
Computer Engineering
The American University in Cairo
mohamedayman15069@aucegypt.edu

Abdelrahman Shaaban
Computer Engineering
The American University in Cairo
abdelrahmanfawzy@aucegypt.edu

1. The Chosen Dataset

After conducting thorough research, we have decided to use the Fakeddit dataset for discernible reasons. Fakeddit consists of over 1 million samples from multiple categories of fake news- 22 different subreddits. It is collected from the website Reddit, a social news, and discussion website where users can post submissions on various subreddits by human experts. Nearly 64% of the dataset is multimodal samples, meaning that Fakeddit has 682,996 multimodal samples having news' headline along with the news piece's attached image. It is worth noting that the multimodal samples are divided into 564000 training examples, 59342 examples for validation, and 59342 testing examples, as shown in figure 1 and table 1. The samples are labeled according to 2-way, 3-way, and 6-way classification categories. In our research project, we use 2-way classification of the text to be ready for usage by different machine learning models explained in class. If time permits, we will be training the associated image of the data, along with making fusion between it and the text to check whether such a model will improve the accuracy of detecting fake news. It means that in this phase, we start preprocessing, cleaning, and extracting features of the text associated with 2-way classification with help of various python libraries. With more details, we can see the features of the dataset itself in figure 2.

Fig. 1: The code shows the shape of the (train,validate, and test).

```
In [3]: FakeNews_dataset_train = pd.read_csv('multimodal_train.tsv', delimiter='\t') #put the data in pd Frame
        FakeNews_dataset_validate= pd.read_csv('multimodal_validate.tsv', delimiter='\t')
        FakeNews_dataset_test= pd.read_csv('multimodal_test_public.tsv', delimiter='\t')

        print(FakeNews_dataset_train.shape)

        print(FakeNews_dataset_validate.shape)

        print(FakeNews_dataset_validate.shape)

        (564000, 16)
        (59342, 16)
        (59342, 16)
```


WOODSTOCK'18, June, 2018, El Paso, Texas USA

https://doi.org/10.1145/1234567890

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 564000 entries, 0 to 563999
Data columns (total 16 columns):
 #   Column                Non-Null Count   Dtype
---  ------                --------------   -----
 0   author                535290 non-null  object
 1   clean_title           564000 non-null  object
 2   created_utc           564000 non-null  float64
 3   domain                396143 non-null  object
 4   hasImage              564000 non-null  bool
 5   id                    564000 non-null  object
 6   image_url             562466 non-null  object
 7   linked_submission_id  167857 non-null  object
 8   num_comments          396143 non-null  float64
 9   score                 564000 non-null  int64
 10  subreddit             564000 non-null  object
 11  title                 564000 non-null  object
 12  upvote_ratio          396143 non-null  float64
 13  2_way_label           564000 non-null  int64
 14  3_way_label           564000 non-null  int64
 15  6_way_label           564000 non-null  int64
dtypes: bool(1), float64(3), int64(4), object(8)
memory usage: 69.4+ MB
```

Fig. 2: Explanation for each column in the dataset.

Dataset Statistics	
Total samples	1,063,106
Fake samples	628,501
True samples	527,049
Multimodal samples	682,996
Subreddits	22
Unique users	358,504
Unique domains	24,203
Timespan	3/19/2008 - 10/24/2019
Mean words per submission	8.27
Mean comments per submission	17.94
Vocabulary size	175,566
Training set size	878,218
Validation set size	92,444
Released test set size	92,444
Unreleased set size	92,444

Table 1: illustration essential information for the dataset.

2. Preprocessing and Cleaning the dataset

Before using some machine learning algorithms to detect the authenticity of news, the data need to be subjected to certain refinements to be valid input to any machine learning models. Some of such refinements are dropping unnecessary columns from the data, lowering case, segmenting the data, removing the punctuations, stopping words, tokenizing, and much more. In this section, detailed information of how we preprocess the data with relevant figures.

2.1 Dropping unnecessary columns from the data.

Fakeddit has 16 columns, 13 of which are considered features, and the other 3 features are the 2-way classification, 3-way classification, and 6-way classification labels. More detailed for the features, they are author name, domain, has image, image url, linked submission id, num comments, score, subreddit, title, upvote ratio, and clean title. We have decided to remove all input features other than clean title for a couple of reasons. First, the author's name, subreddit, and domain column are meaningless, as they do not have any additional information to identify the authenticity of the news. Upvote ratio, score, num comments are misleading in the datasets, because there are several fake examples having high scores and upvote ratio. Regarding linked submission id, we have found that all of them are NaN. We dropped the title column since clean title is the same as title other than the fact that clean title is cleaned from hashtags and URLs. Last two features the image and its URL are dropped since we do not deal with images for now. However, we added the word count and stop words count for further preprocessing as shown in table 2 below.

Out[6]:

	clean_title	2_way_label	word_count	stop_words_count
0	my walgreens offbrand mucinex was engraved wit...	1	15	7
1	this concerned sink with a tiny hat	0	7	3
2	hackers leak emails from uae ambassador to us	1	8	3
3	puppy taking in the view	1	5	2
4	i found a face in my sheet music too	0	9	5

Table 2. : The data frame after dropping non-essential features.

2.2 Removing Punctuations, Lowering the word cases and Expanding contracted characters.

After dropping the unnecessary columns, we start removing all unnecessary punctuations, lowering the word cases, and expanding the contracted characters, like 'm to am, to be standardized for being prepared for more advanced preprocessing as shown in figure 3.

Fig. 3: The code illustrating how we remove punctuations and expand contracted characters.

2.3 Removing Stop Words.

Stop words cause catastrophe for the data since their frequencies in our data are quite high compared to the other important words. In this sense, such insignificant words create noise when used as features in text classification. Prepositions, conjunctions, articles, and some pronouns are stop words. In our data, we removed all common words such as, a, about, an, are, as, at, be, by, for, from, how, in, is, of, on, or, that, the, these, this, too, was, what, when, where, who, will, etc. After removing the stop words using "spacy.lang.en.stop_words", we have the processed data without stop words for the next preprocessing step as shown in the figure 3.

2.4 Removing non-standard language.

After deleting the stop words, we need to check whether the words make sense or not. From the figure number 3, we removed all non-standard English language words, like aaaaa, to avoid any noise coming from them. The main rationale for thinking of removing non-standard words is that stemming does not return the non-standard language to its root, so we need to use this step to remove the unnecessary words as shown in figure 4.

Fig. 4: Code illustrates how to remove non-standard language

2.5 Removing Rare and Common Words.

Before removing the common and least frequent words, we deleted any sentence whose length is less than three. With a help of figure 5, coming to the next step, we needed to remove the high-frequent words, as they are common between many samples and they are slightly useless when it comes to the training process, as it hinders the effect of average words. Similarly, we also needed to remove the rare words, as it makes noise when coming to the training process. In this sense, we made a threshold for rare and common words. For common words, we deleted the first ten of the most frequent words. Also, we removed the least frequent words whose threshold frequency is less than 100 words for some reason- I will mention details in the feature analysis section. They are nearly five thousand words.

After doing this step, we removed the sentences whose lengths are between 1 and 2.

```
In [24]: corpus= temp_df['clean_title'].apply(lambda x: ' '.join(x))
         all_text= ' '.join(corpus)
         words= all_text.split()
         freq_com=pd.Series(words).value_counts()
         freq_com
         Most_common=freq_com[:10]
         rare= freq_com[freq_com==1]
         freq_com

Out[24]: like          16718
         man           14145
         nt            13563
         find          12453
         new           11055
                        ...
         teratoma          1
         unshak            1
         needlework        1
         ocular            1
         mot               1
         Length: 20231, dtype: int64
```

Fig.5: The frequency of each word.

2.6 Base the Word to the Root.

```
In [18]: def tokenization(x):
             return x.split()

         df_train['clean_title']= df_train['clean_title'].apply(lambda x: tokenization(x))

         df_train.head()
```

Out[18]:

	clean_title	2_way_label	word_count
0	[engraved, different, order]	1	15
1	[concerned, sink, tiny, hat]	0	7
2	[leak, ambassador]	1	8
3	[puppy, taking, view]	1	5
4	[found, face, sheet, music]	0	9

Fig. 6: The code used to tokenize the text.

After tokenizing the processed data as shown in figure 6, the next step is to transform all tokenized words to a standard form (its root). In other words, the words need to be returned into their original form. For instance, the original word of “playing”, “plays”, and “play” is simply “play”. This will help us decrease the number of word types for the sake of avoiding the out of memory problem. As a result, the classification will be much faster and efficient. We use the porter stemmer from NLTK library. However, the problem of stemming may lead to having a word like plai instead of play, as it removes only the suffix without considering the authenticity of the word, so we use lemmatize from library “spacy” to have the right word as shown in figure 7.

```
In [19]: from nltk.stem.porter import PorterStemmer
         porter = PorterStemmer()
         def stemming(x):
             stems = []
             for t in x:
                 lemma=str(porter.stem(t))
                 if lemma == '-PRON-' or lemma == 'be':
                     lemma=t
                 stems.append(lemma)
             return stems
         df_train['clean_title']= df_train['clean_title'].apply(lambda x: stemming(x))
         df_train.head()
```

Out[19]:

	clean_title	2_way_label	word_count
0	[engrav, differ, order]	1	15
1	[concern, sink, tini, hat]	0	7
2	[leak, ambassador]	1	8
3	[puppi, take, view]	1	5
4	[found, face, sheet, music]	0	9

Fig.7: code for stemming each word

2.7 Storing the data using Sparse Matrix.

Unfortunately, we have tried to use a bag of words where each word has nearly 27 thousand entries in such an array, leading to a need for nearly 94 GB Ram to load. This is completely inefficient in our case. In this sense, we use the idea of a sparse matrix. More detailed, the dictionary has for each sentence the number of words with their frequencies, as shown in figure 8. If any word frequency is zero, there is no need for storing it as shown in the figure below. This has saved a lot of space complexity on average, theoretically.

```
         a={}
         for t in x:
             a[t]= x.count(t)
         return a
temp_df['dic_words']=temp_df['clean_title'].apply(lambda x: make_dic(x))
temp_df.head()
```

41]:

	clean_title	2_way_label	word_count	dic_words
0	[engrav, differ, order]	1	3	{'engrav': 1, 'differ': 1, 'order': 1}
1	[concern, sink, tini, hat]	0	4	{'concern': 1, 'sink': 1, 'tini': 1, 'hat': 1}
2	[leak, ambassador]	1	2	{'leak': 1, 'ambassador': 1}
3	[puppi, take, view]	1	3	{'puppi': 1, 'take': 1, 'view': 1}
4	[face, sheet, music]	0	3	{'face': 1, 'sheet': 1, 'music': 1}

Fig. 8: The sparse Matrix in dic_words column

3. A Full Analysis of the different Features

3.1 Label Distribution.

The labels are mostly equally distributed between the two classes 0, for the fake news class, and 1, for the true news class. By going through all the instances after the data preprocessing, it is around 231,723 samples that belong to class 0 and 207,679 samples that belong to class 1. This class distribution is shown below.

```
In [5]: df_train=df_train[['clean_title','2_way_label']]
        print(df_train['2_way_label'].value_counts())
        df_train.head()
```

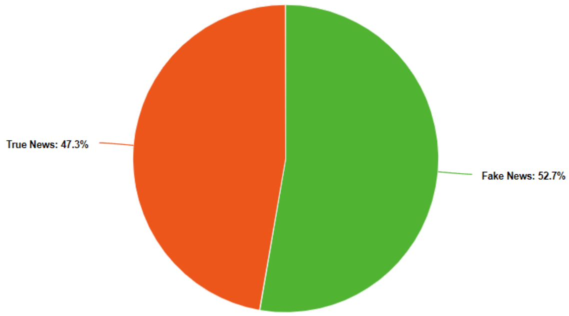

Fig 9: Label Distribution between the fake and authentic information.

3.2 Feature Extraction.

By preprocessing the dataset, we have, dropping the nans and the useless rows with one-or-two-word titles, besides other refinements such as lower casing, stop words removal, punctuation removal, tokenization, stemming, and lemmatization, it helped in reducing the size of the actual, useful data, and making the learning process much more efficient, affected only with the most relevant features. It was able to reduce the total bag of words from 121,000 words to around 20,000 words. That size of the feature set is the number of all these words after removing the most (10) and the least (100) frequent words, which is around 2800 words.

3.3 Instances' Distribution.

We measured the instances' distribution based on the words' frequencies in the whole text dataset, and represented this distribution using bar graphs, obtaining the most and least frequent words. We mainly measured the frequency distribution of the most frequent 1000 words, the least frequent 1000 words, in addition to the distribution of 100 random words. In addition, we made a word cloud of all words' frequencies as shown in figure 13. All of them are shown below. More detailed, fig 10 shows the most 1000 frequent words. While figure 11 illustrates the least frequent words (frequency is 1). Figure 12 is the frequency for 100 randomized values.

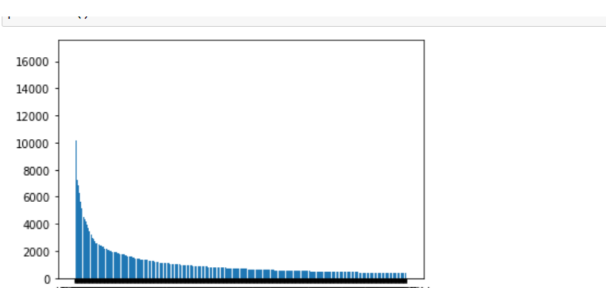

Fig. 10: The most frequent 1000 words.

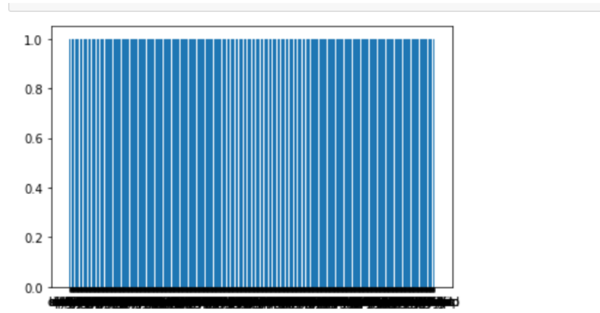

Fig. 11: The least frequent words (frequency = 1).

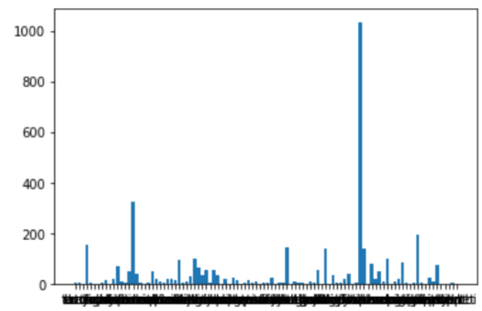

Fig. 12: the frequency of 100 random values.

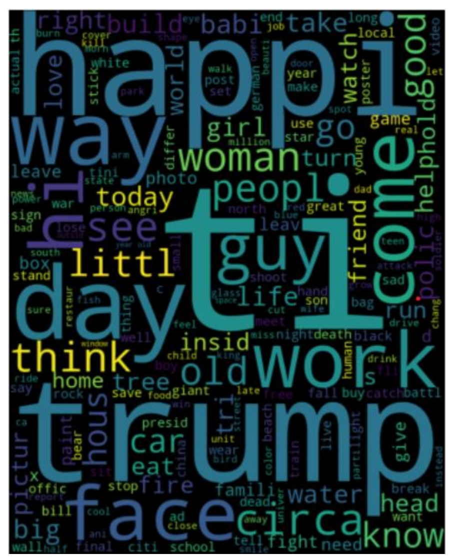

Fig. 13: Word cloud for the preprocessed dataset.

3.4 Feature-label Correlation.

We needed to measure the correlation between all the remaining words and the labels to obtain how useful each word was in the training process and how much it can contribute for classification. The correlation was measured using stacked bar charts that show how each word

correlated with the two labels. Besides measuring the correlation between all the words and the two labels as shown in figure 17, we measured it for different groups of words, which are the most frequent ten words as shown in figure 14, and the least frequent with a frequency of less than 10 as shown in figure 15 and with least frequency 100 as shown in figure 16.

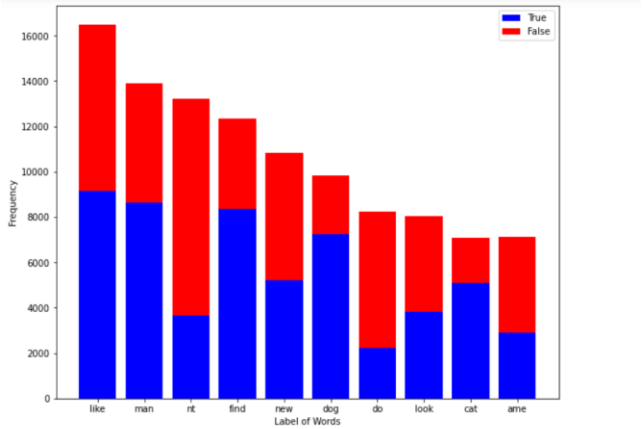

Fig. 14: The correlation for the most frequent 10 words.

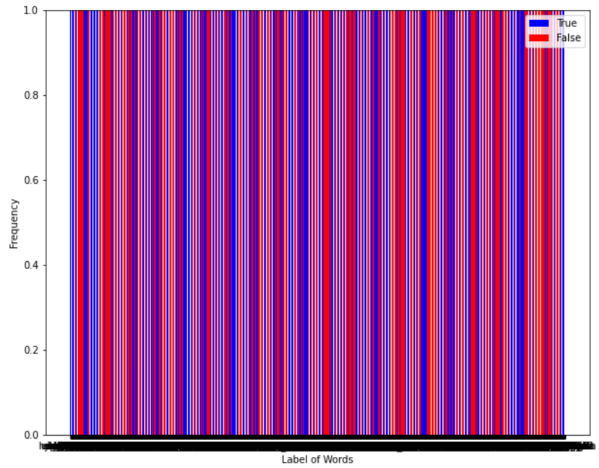

Fig. 15: The correlation for the words whose frequency less than 10.

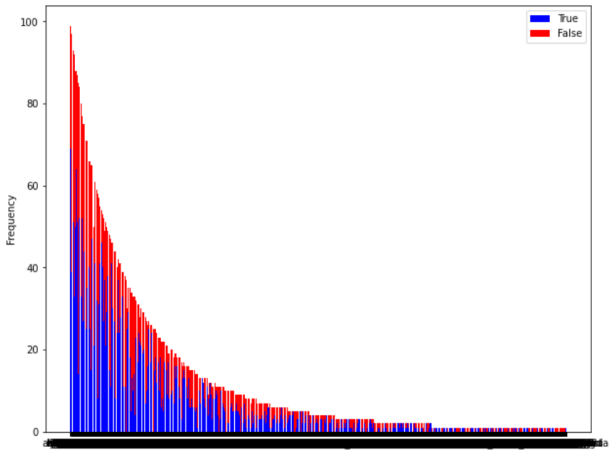

Fig. 16: The correlation for the least frequent 100 words.

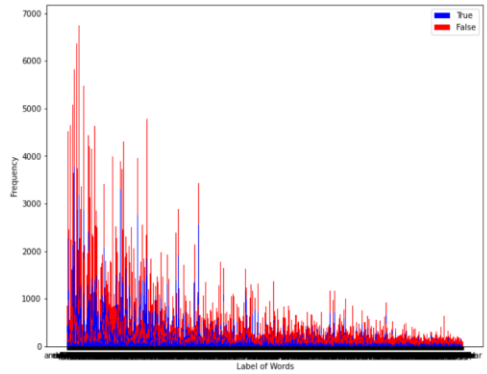

Fig. 17: The correlation for the whole unique words.

In addition, we computed the TF IDF by computing the TF and IDF for the whole dataset to measure how relevant each word is to its news' title as shown in figures 18, 19, and 20.

```
In [22]: TF = []
         def calculate(x):
             sent_TF={}
             sent_len=len(x)
           # print(type(x))
             for (key,value) in x.items():
                 sent_TF[key]= value/sent_len
             TF.append(sent_TF)
         temp_df['dic_words'].apply(lambda x: calculate(x))
         TF

Out[22]: [{'differ': 0.5, 'order': 0.5},
          {'concern': 0.25, 'sink': 0.25, 'tini': 0.25, 'hat': 0.25},
          {'leak': 0.5, 'ambassador': 0.5},
          {'puppi': 0.3333333333333333,
           'take': 0.3333333333333333,
           'view': 0.3333333333333333},
          {'face': 0.3333333333333333,
           'sheet': 0.3333333333333333,
           'music': 0.3333333333333333},
          {'bride': 0.16666666666666666,
           'exchang': 0.16666666666666666,
           'fatal': 0.16666666666666666,
           'shoot': 0.16666666666666666,
           'we': 0.16666666666666666,
           'd': 0.16666666666666666},
          {'meat': 0.25, 'pig': 0.25, 'eat': 0.25, 'milk': 0.25},
          {'convert': 0.2, 'local': 0.2, 'teen': 0.2, 'circa': 0.2, 'ad': 0.2},
          {'victori': 0.3333333333333333,
```

Fig 18: The Code to get TF.

```
In [23]: # TFIDF
         TF_IDF_list=[]
         for t in TF:
             TF_IDF  = {}
             for (key,value) in t.items():
                 TF_IDF[key]= value*IDF[key]
             TF_IDF_list.append(TF_IDF)
         TF_IDF_list

Out[23]: [{'differ': 2.808921147032861, 'order': 3.0580384762316},
          {'concern': 1.842089701189932,
           'sink': 1.6141982556535566,
           'tini': 1.394993005002578,
           'hat': 1.5786261041832778},
          {'leak': 4.185371530666642, 'ambassador': 4.108951928551918},
          {'puppi': 2.1545273049826372,
           'take': 1.5492722687180107,
           'view': 2.0612548250903675},
          {'face': 1.5193952154910417,
           'sheet': 2.6860482125389895,
           'music': 2.07527722209512},
          {'bride': 1.3016362525859662,
           'exchang': 1.3408018511245845,
           'fatal': 1.2451839938419285,
           'shoot': 0.8858930059681344,
           'we': 1.0119724071780911,
           'd': 0.8758990951735415},
          {'meat': 1.7733458458942655,
```

Fig 19: The Code to get TF/IDF.

```
In [62]: import math
         IDF = {}
         doc_length = len(df_train['clean_title'])
         for w in freq_dict:
             IDF[w] = math.log(doc_length/(freq_dict[w]+1))
         plt.bar(*zip(*IDF.items()))
         plt.show()

C:\Python\Python39\lib\site-packages\matplotlib\backends\backend_agg.py:240:
  font.set_text(s, 0.0, flags=flags)
C:\Python\Python39\lib\site-packages\matplotlib\backends\backend_agg.py:203:
  font.set_text(s, 0, flags=flags)
```

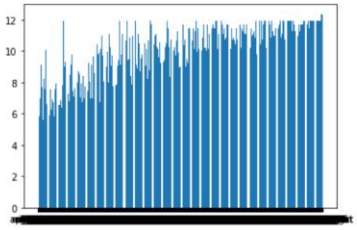

Fig 20: The correlation between words and IDF.

4. Final list of Chosen Features

From the correlation curves, specifically graphs 14, 15,16, and 17, above and the computed TF-IDF in figure 20, we can obtain how it is very useless to use words with a frequency of less than 100, as the correlation is going down as the frequency goes down. In figure 15, we can notice how noisy the words with frequency less than 10 are. Then, we decided to plot the least 100 frequent words as shown in figure 16. We noticed that there is a clear correlation between the label and the words. Similarly, the most frequent ten words have the same scenario. Therefore, we dropped all these words, and we will start training the models by using words with a frequency of more than 100, which are around 2800 words. We believe that these are the most influential words to create a clear pattern when entering as an input for different machine learning models. For future phases, we have been thinking of using text embedding to have a more stronger correlation between words.