**Fake News Detection**

# **Proposal**

It is widely believed that "fake news," broadly defined as false or misleading content that poses as news, is pervasive online and substantially negatively impacts democracy. The issue is that numerous news sources are using various media, which is problematic because it can be challenging to tell whether a news story is trustworthy. Therefore, we wish to develop a model to ascertain the veracity of news articles to address this problem.

According to [Jennifer Allen](https://www.science.org/doi/10.1126/sciadv.aay3539), the deliberate dissemination of disinformation online since the 2016 U.S. presidential election, especially on social media platforms like Twitter and Facebook, has sparked interest across multiple disciplines. Additionally, towards the start of the pandemic in 2020, fake news resurfaced as a major issue and had a significant global impact. Since the internet allows people to freely share thoughts and information while simultaneously serving as a source of information that people utilize to acquire news and knowledge, spotting fake news is crucial. According to [Deloitte](https://www2.deloitte.com/us/en/insights/industry/technology/study-shows-news-consumers-consider-fake-news-a-big-problem.html), more than 70% of US news consumers are concerned about fake news, which means that fake news is a big issue that the modern world has to face nowadays. Additionally, anyone that accesses the news or the general public is impacted by this issue. Therefore, developing a model and user-friendly tool to identify fake news is anticipated to help address this serious issue.

There is no denying the adverse effects of fake news, and many people are affected by this issue, which has the potential to impact their everyday lives significantly. Online "fake news" is frequently widely circulated in the US. In 2016, a shooting incident in Washington, D.C., was brought on by the spread of incorrect information, according to [The Washington Post](https://www.washingtonpost.com/news/local/wp/2016/12/04/d-c-police-respond-to-report-of-a-man-with-a-gun-at-comet-ping-pong-restaurant/). Fake news has an impact by changing how people view crucial issues and topics and by altering facts, realities, and beliefs. Cultural values are also being impacted more and more by it. Therefore, fake news is a serious concern since it can be difficult for individuals to determine if an emergency or situation is real or not if the news is untrustworthy.

The public is not always aware of how to verify the news, and there are times when it may be challenging to do so. Therefore, it is essential to encourage the development of technologies that can detect false news and for the major media to release accurate and genuine information in order to battle it. Therefore, attempts were made to automate and optimize fake news detection. Our False News Detection project aims to develop a tool to identify fake news via pattern-finding machine learning and natural language processing on news sources and articles.

Our goal for this project is to (1) decrease the number of unreliable news articles being shared, (2) give users more confidence in the reliability of the news articles they read by providing a reliability score, and (3) deter users from clicking on news articles that are fake by placing a warning of reliability next to the link.

# **Introduction**

We are investigating natural language processing for this report, a comprehensive study of how computers and machines can comprehend human-to-human communication and how machines process texts based on contextual data. Specifically, we classify news pieces as "real" versus "fake" news, a binary classification problem using natural language processing. Fake news is misleading material disseminated under the pretext of a legitimate news story. In our project, we have employed different models to classify text into different classes:

* Decision Tree Classifier
* Logistic Regression
* Random Forest Classifier
* Gradient Boosting Classifier
* Support-Vector Machines (SVM)
* Multinomial Naive Bayes
* Neural Network with LSTM

We divided the solution into three stages: preprocessing, text-to-numeric representation conversion with previously taught methods, and model evaluation with cutting-edge machine learning techniques. We first analyzed the data set, paying close attention to the text portion that describes how the data is distributed, and then we turned each text into a numeric representation using pre-training models for vector representation such as TF-IDF Vectorizer, Count Vectorizer, and Word2Vec Vectorizer. Finally, we classified our numerical conversion data using essential machine learning techniques, including neural networks and classification algorithms.

# **Dataset**

We used the WELFake dataset from Kaggle. This dataset consisted of 72,134 instances of real and fake news articles. 35,028 of the instances were real news articles, and 37,106 of the instances were fake news articles. This dataset was made up of four attributes: serial number, title, text, and label. The serial number was a unique identification number for each instance, the title was the article's heading, the text was the body of the article, and the label identified the article as being real or fake. The real articles were labeled as 1, and the fake articles were labeled as 0. As we moved forward with exploratory data analysis and modeling, we focused exclusively on the text attribute since we anticipated that the language and terms utilized in the text would be crucial to differentiate the samples of real and fake news.

Some issues with this dataset are that links or sources were not provided with each instance, we do not know how they classified each news article as real or fake, and we do not know the full extent of how they curated this dataset. By not having the links or sources for each article, we are unable to double-validate whether the article was real or fake. Additionally, we cannot determine whether there was any bias in the results by not knowing how they classified each news article. Lastly, since we do not understand how they curated this dataset, we do not know the full extent of sources or how they collected the data.

# 

# **Pipeline**

# 

# Figure 1

# The pipeline of this project is shown in Figure 1. In this, we started with the data, cleaned it, extracted features, performed the exploratory data analysis, then selected important features for modeling, which ended up being the text attribute. After this, we split the dataset into training data and testing data. We then took the training data to train the models. We then moved into the model evaluation and took the trained model to retrain the model if needed. This was repeated as many times as necessary before the articles were finally classified.

# **Data Pre-Processing**

## **Cleaning/Denoising Text**

To clean the text, we eliminated all non-textual characters (non-alphabetic characters like punctuation, additional delimiters, etc.). We first used regex to eliminate symbols, punctuation, letters, and URLs before creating a function to clean the data. Then, we checked that each character was in the correct format using the Unicode library. Then, we eliminated all terms from the stopword corpus using nltk's corpus.stopword library. The most frequent words in a language, such as "a," "be," "very," "should," etc., are called stop words. They frequently offer nothing to the content and are meaningless. They appear in text the most frequently as well. Therefore, we assumed that eliminating stop words could have several benefits. Since we are only interested in finding fake news in English, we continued making all of the text lowercase and removing any rows containing characters from languages other than English. We developed another function after this one to convert the empty instances into null values, and then we deleted all the null values.

## **Lemmatization**

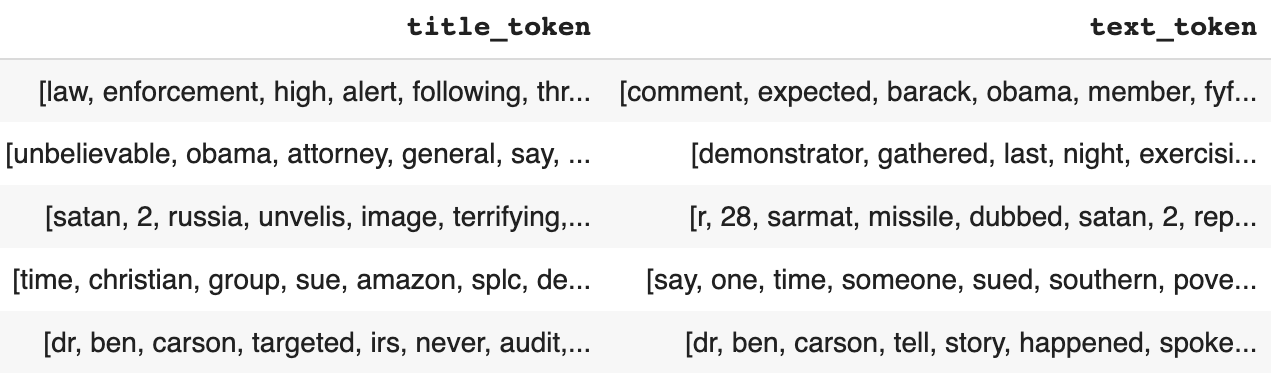
Lemmatization aims to condense a word's inflectional forms and is occasionally derivatively related to a basic format used frequently. While lemmatization is virtually identical to stemming in that it always returns the chopped term with some dictionary definition, stemming cuts off the provided word to its base word. Since stemming does not care if a word has a meaning or not, it will just cut and return a word, the word "easy," for instance, will be replaced as "easi," which has no dictionary definition. Lemmatization, however, is concerned with the meaning of the term that it returns. Because lemmatization algorithms consult the dictionary to create a correct root word from the given word, the new word will always be meaningful. Using text blobs Word library, we could lemmatize each word into its root. This is significant because it enables us to group the terms together for additional examination.



# Figure 2

## **Tokenization**

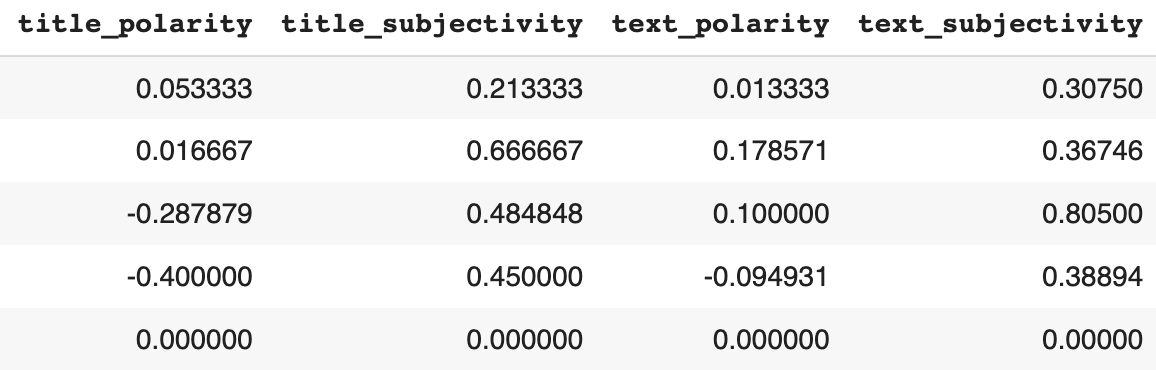
Another common task in natural language processing is tokenization. Tokenization—the division of a text block into smaller, discrete units—is an essential step. Here, we assume that space serves as a delimiter and creates tokens based on space. Tokenization is significant because it makes it simple to interpret the text's meaning by looking at its vocabulary. In our project, we created a tokenization function using the word function from the text blob library, which we then applied to new columns of the tokenized text and title. Later, the tokenization could be looked into for further analysis.



# Figure 3

## **Sentiment analysis**

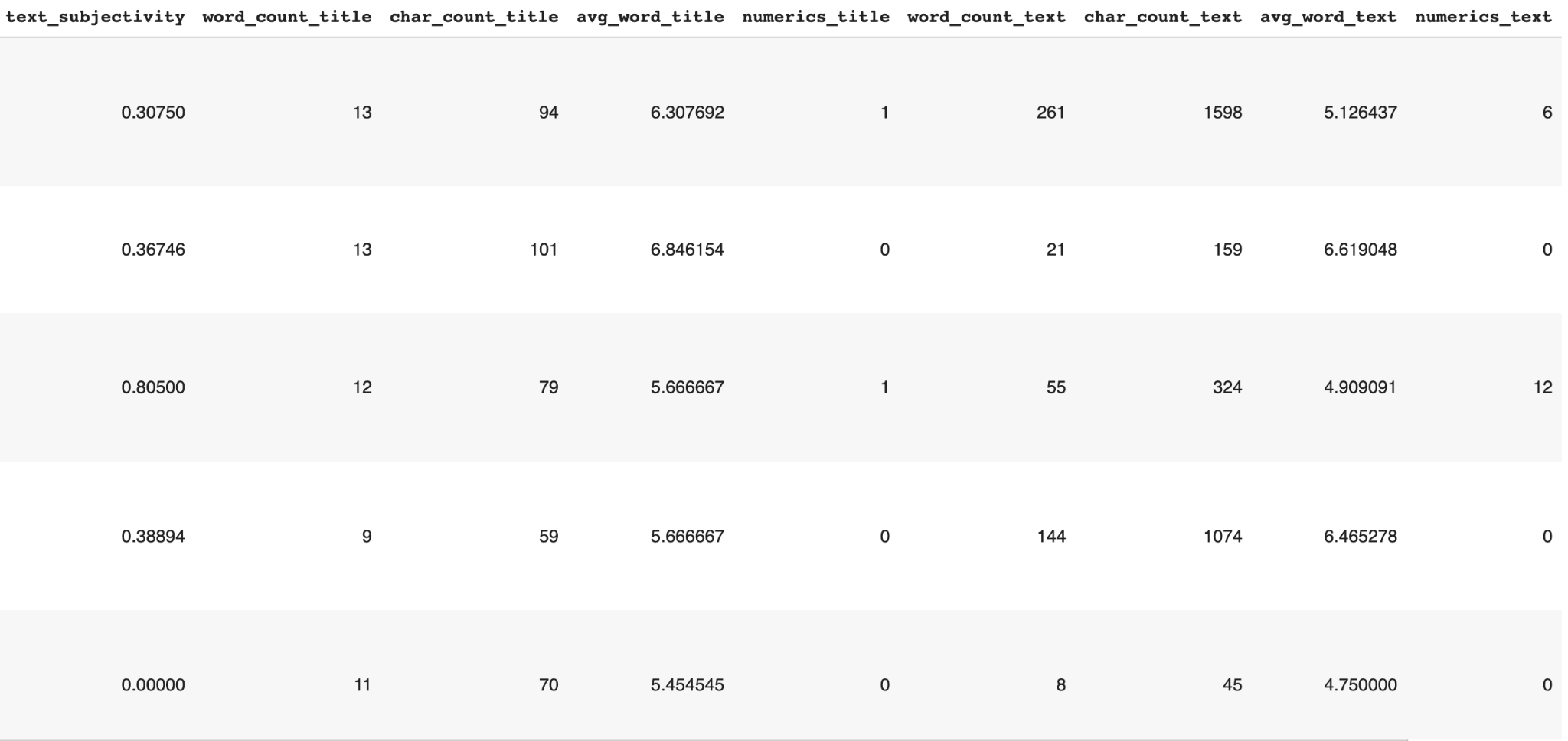
The process of identifying a positive or negative sentiment in the text is known as sentiment analysis. Sentiment analysis is quickly becoming a crucial tool to monitor and comprehend sentiment in all forms of data because people express their views and feelings more freely than ever before. For sentiment analysis, we gathered the subjectivity and polarity of each text and title instance using the text blob library's sentiment function. These could be investigated for potential utility as a predictive feature.



# Figure 4

## **Basic Feature Extraction**

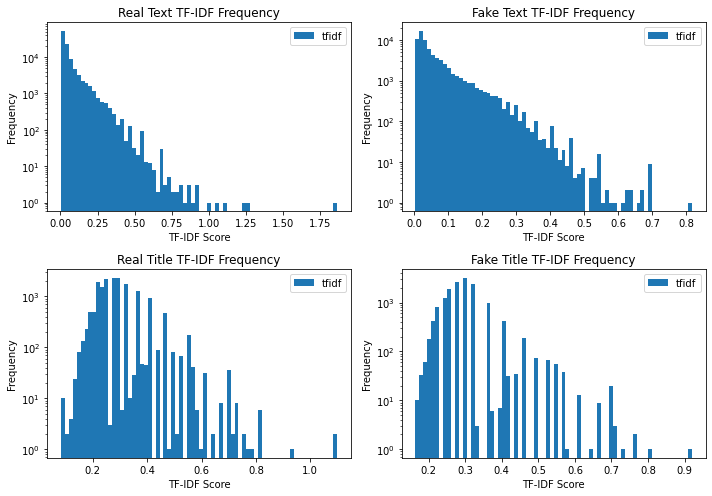
## For the basic feature extraction, we found the number of words, characters, numerics, and the average number of words. These data are used in the EDA part to analyze the data set through histograms and correlation. The basic feature extraction could also be considered a possible predictive feature.



# Figure 5

## **TF-IDF**

For the TF-IDF, we started by separating the dataset based on whether or not the articles were real or fake. This way, further analysis could be done later to determine if TF-IDF would have predictive properties. After we separated the datasets, we created dictionaries with the count of each word for both of the new datasets. Then, we created a function to calculate the term frequency, inverse document frequency, and term frequency-inverse document frequency. We then applied the function to the title and text of both the real and fake datasets. We found that for the real text, the TF-IDF was mostly between 0 and 1 but went just past 1.75, and for the fake text, the TF-IDF was mostly between 0 and 0.7 but went just past 0.8. For the real and fake title TFIDF, most of the values stay between 0 and 0.8; however, the real title goes past 1, and the fake title goes only a little past 0.9. These findings can be seen in Figure 6.



# Figure 6

# **Exploratory Data Analysis**

# **Data Summary**

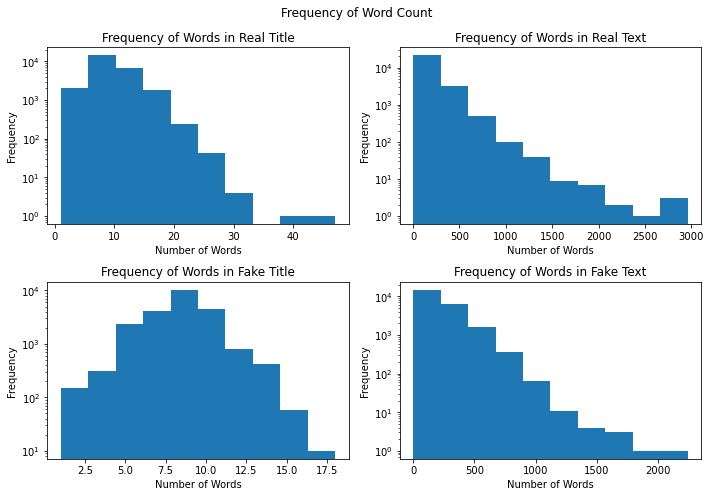
For the data summary, we collected the number of instances in the dataset and the number of real and fake instances and created a histogram of real versus fake instances. Since this was completed after the pre-processing, the total number of instances dropped to 48,390, with 52.4% being real and 47.6% being fake. The histogram can be seen in Figure 7.



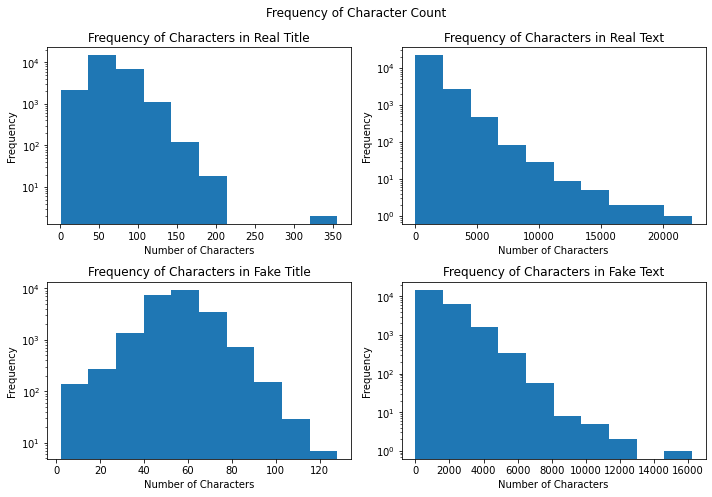
# Figure 7

# **Number of Words/Characters/Numerics**

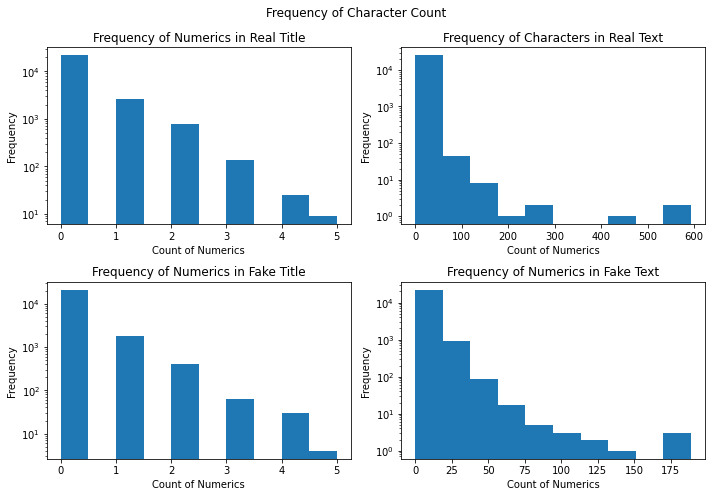
For the number of words/characters/numerics, we created histograms to see the differences between the real and fake instances for each attribute. These plots can be seen in Figures 8-10. These show that the frequency of words, characters, and numerics was higher for real text and titles.



# Figure 8



# Figure 9



# Figure 10

# **Wordcloud**

We used the Wordcloud function from the wordcloud library for the word cloud. First, we set the max words to 2000, the width 1600, the height to 800, and stopwords to the STOPWORDS library from the wordcloud library. We then showed the wordcloud using matplotlib.pyplot.imshow, and set the interpolation to bilinear. The wordcloud plots can be seen in Figure 11. In these plots, we can see that in the text, "united state," "donald trump," and "said" were very common for both real and fake news. However, we can see that in the real text, "hillary clinton" and "people" were more common, whereas in the fake text, "country" and "one" were more common. In the title plots, we can see that "trump" is common in both real and fake. However, we can see in the real titles that "video" and "trump" are common, and in the fake titles, we can see that "new york" and "york time" are common.

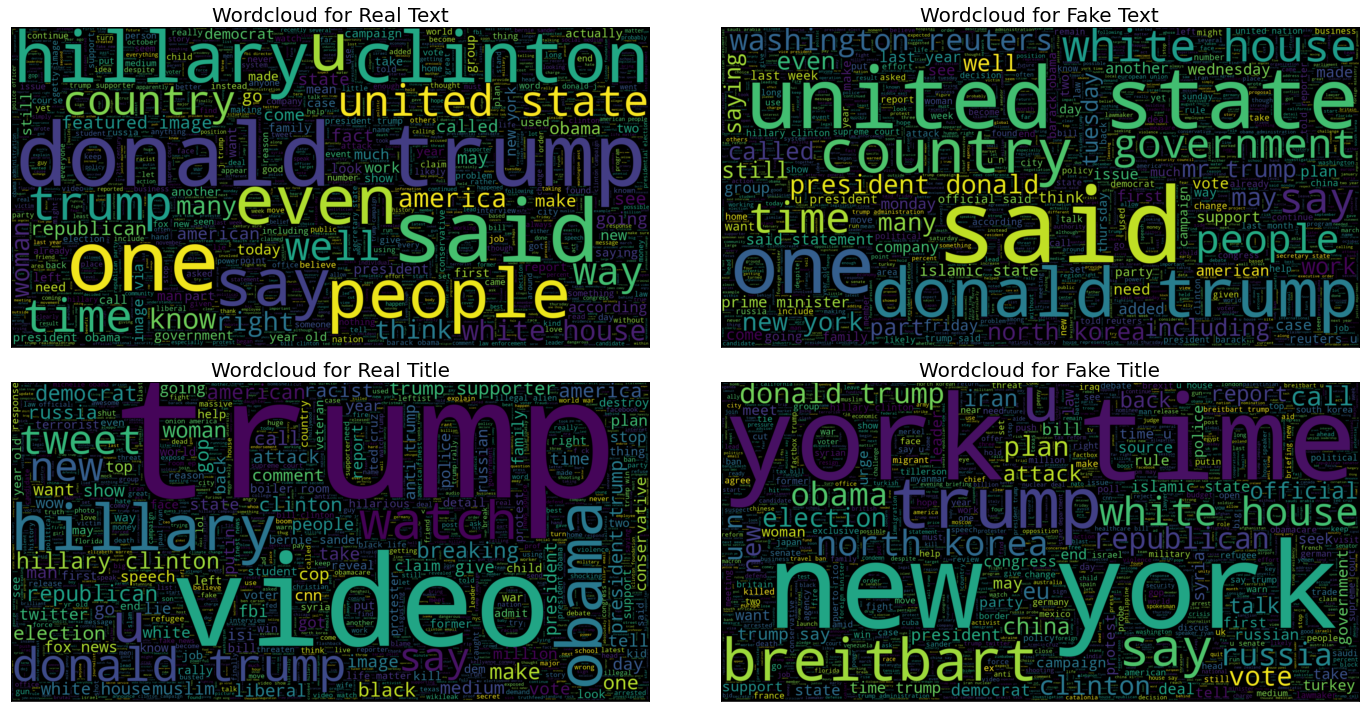


Figure 11

# **Ngrams Analysis**

For ngrams analysis, we created a function to gather the top ngrams using sklearn.feature\_extraction.text.CountVectorizer. We then plotted the most common bigrams and trigrams for real and fake titles and text. Figure 12 shows bigrams, and Figure 13 shows trigrams. For real text and text bigrams, donald trump and hillary clinton were the most common. For fake text bigrams, united state and donald trump were the most common. For fake title bigrams, new york and york time were the most common. For real text trigrams, featured image via and donald trump realdonaldtrump were the most common. For real title trigrams, black life matter and boiler room ep were the most common. For fake text trigrams, president donald trump and president barack obama were the most common. For fake title trigrams, new york time and briefing new york were the most common.

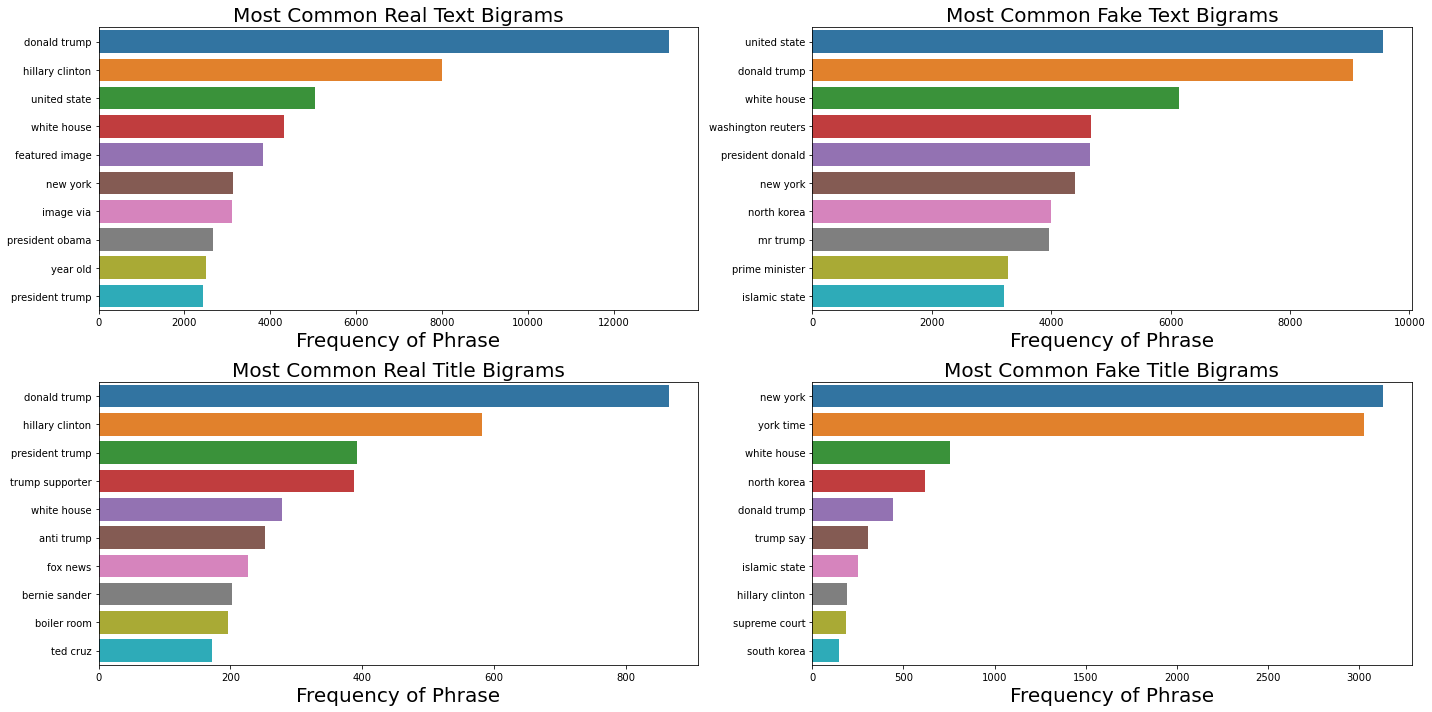


Figure 12

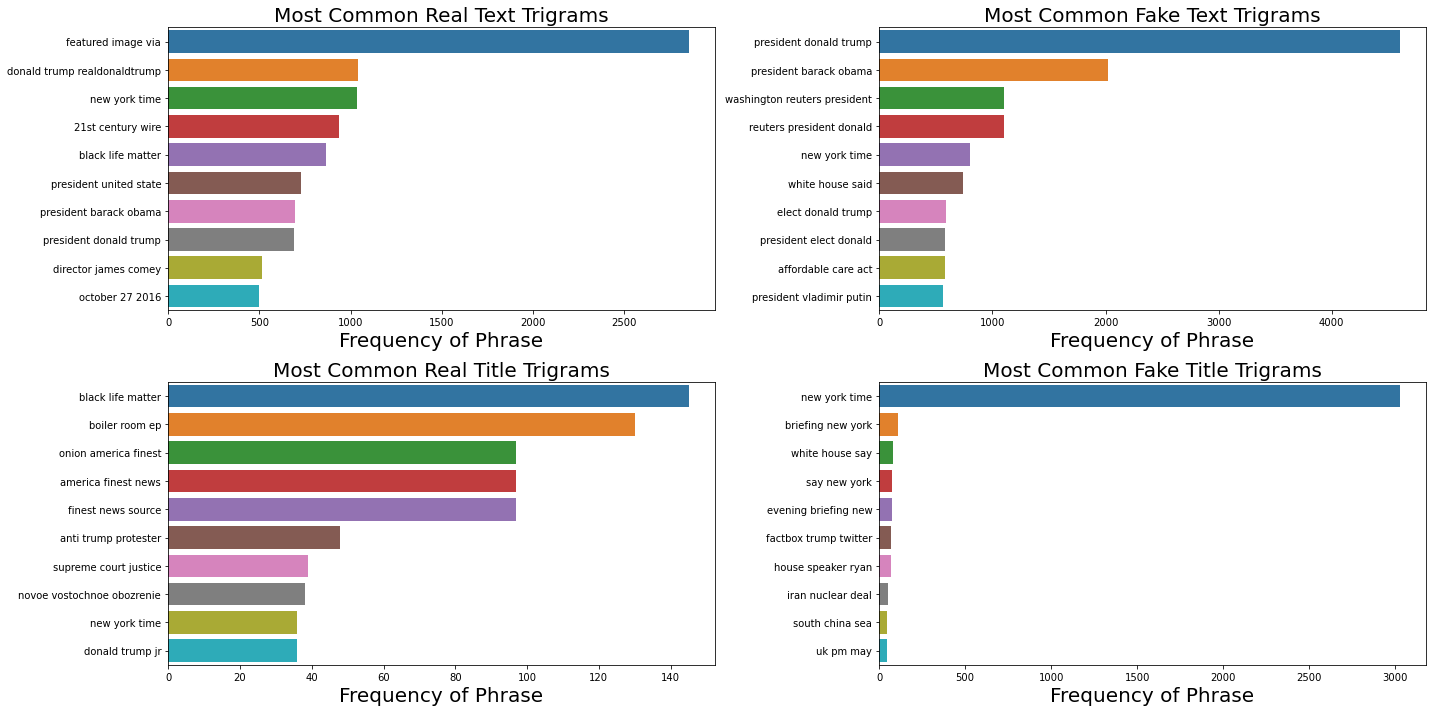


Figure 13

# **Compare F1**

When we compared F1 scores, we used sklearn.model\_selection.train\_test\_split, sklearn.feature\_extraction.text.CountVectorizer, sklearn.naive\_bayes.MultinomialNB, and sklearn.metrics.f1\_score. For train\_test\_split, we set the test size to 25% and the random state to 42. In this, we found that the models with an n-gram of 1, 1 was 0.87, models with an n-gram of 1, 2 was 0.91, models with an n-gram of 1, 3 was 0.92, and models with an n-gram of 1, 4 was 0.92. Since the F1 score stopped increasing after an n-gram of 1, 3, we decided to use bigrams and trigrams.

# **Topic Modeling**

To complete the topic modeling, we used the gensim and spacy libraries. We then created a function with these libraries. For gensim, we used the gensim.models.ldamodel.LdaModel, and set the number of topics to 20, passes to 10, random state to 0, and iterations to 50. We did this for both real and fake title and text attributes. For the real title, the first topic contained "election," "u," "trump," "death," "government," "could," "find," "deal," "nuclear," and "post." For the fake title, the first topic contained "trump," "protest," "breitbart," "anti," "military," "head," "face," "speech," "despite," and "protester." For the real text topics, the first topic contained "wire," "21st," "century," "21wire," "radio," "room," "pm," "boiler," "acr," and "u." For the fake text topics, the first topic contained "farmer," "south," "brazil," "water," "sudan," "food," "kiir," "farc," "perdue," and "farm."The topic modeling was performed to determine if there was any sort of predictive power, which we could not find.

# **Most/Least Common Words**

For most and least common words, we used the collections.Counter function. We then used this to create a function to gather the ten least and ten most common words. By determining the most and least common words, we were able to figure out any extra words that needed to be removed from the dataset, as well as if we had missed any additional words that needed to be removed.

# **N-grams**

Ngrams are essential since they help to represent the structure of the language. For ngrams, we focused primarily on bigrams, or sets of 2 words, in the dataset. However, we also created trigrams, or sets of 3 words, for the dataset. To calculate the ngrams, we used the TextBlob.ngrams function. This collected the trigrams and bigrams of the text and title attributes. We then double-checked that the stopwords and punctuation were removed after gathering the ngrams.

# **Correlation**

# We calculated the correlation to try and determine if the label attribute had a strong correlation with any of the other attributes. While none of the attributes had a strong correlation with the label, the label attribute had a weak positive correlation with character count title, word count title, and text subjectivity.

# **Modeling**

# **Natural Language Processing Models - Pre-Training Algorithm**

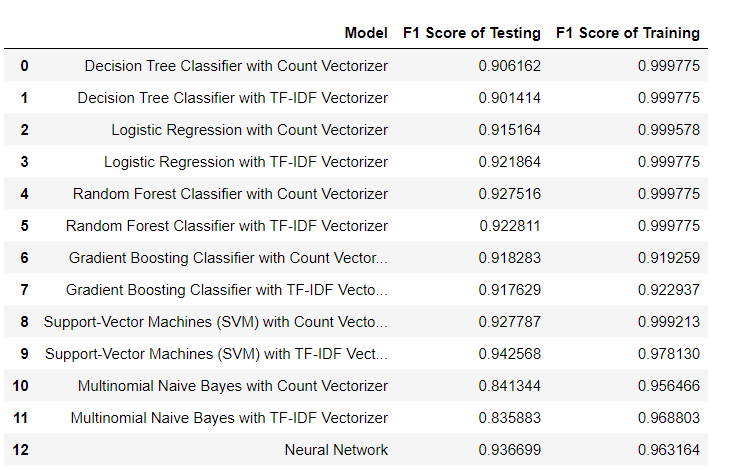
Following text cleaning, two pre-training algorithms - Count Vectorizer and TF-IDF Vectorizer - are used to convert the textual input into numerical representations in the form of vectors. Except for the neural network, we utilized both vectorizers to determine which was superior. Every sample, initially made up entirely of text, is transformed into a vector of features.

The Count Vectorizer provides a simple approach for tokenizing a collection of text documents, building a vocabulary of recognized words, and using that vocabulary to encode new documents. The TF-IDF Vectorizer must be computed along with the inverse document frequency and product of term frequency. As the name implies, the TF-IDF calculates values for each word in a document by dividing the frequency of the word by the proportion of texts in which it appears.

In addition to the Count Vectorizer and TF-IDF Vectorizer pre-training algorithm, we also use the Word2Vec Vectorizer in our Neural Network model. Word2Vec is another cutting-edge model that converts words into vectors. A straightforward neural network called Word2Vec merely attempts to predict the following word within a context given a sequence of words. Word2Vec uses two different architectures: (1) Continuous Bag of Words (CBOW), which predicts the current word based on the context of words in a specified window (2) Skip Gram, which, given a current word, predicts the words in the surrounding context within a given window. Word2Vec essentially expresses a vector for each word in the context as a set of weights for a specific link from a node in the input layer to a neuron in the hidden layer.

# **Different Machine Learning Models - 70/30 Split versus 80/20 Split**

Firstly, we divided our dataset into 70% for training and 30% for testing. Even though all models' accuracy and F1 scores are consistently high, their training and testing test performances have a big gap. The generalization performance of all models was recorded in the table below using the F1 score.



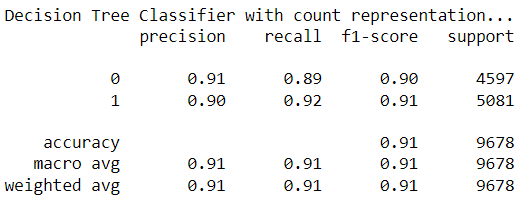
According to the table, the training performance of several models is much higher than on the testing data, indicating that the models are overfitting. Overfitting happens when a model learns the information and noise in the training data to the point where it adversely affects the model's performance on testing data. This indicates that the machine understands concepts from the noise or random oscillations in the training data. However, these ideas do not apply to testing data, which poses a difficulty for the models' capacity to generalize.

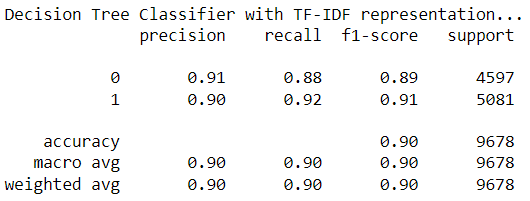
To overcome overfitting, we consider different methods, such as using k-fold cross-validation, using the additional validation dataset for fine-tuning, or increasing the size of the training dataset. However, since we lack time to run the fine-tuning hyperparameters and cross-validation, we decided to change our training and testing dataset ratio to 70% and 30%. Furthermore, by providing algorithms with more data, we expect that it will help improve their ability to recognize signals. Hence, we decided to use the 80/20 split for our machine learning models.

* 1. **Machine Learning Models**

# **Decision Tree Classifier**

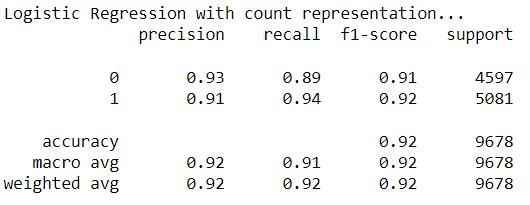
For the Decision Tree Classifier, we used sklearn.DecisionTreeClassifier function. This is a supervised classification model that classifies based upon decision rules from the features. Essentially, the model will look at each instance in the dataset and pass it through a flowchart to classify it. This model is beneficial since it typically requires little data preparation and typically performs well. The disadvantages of this model are that overfitting can occur, and it can become biased if one class is more common than the other. The results of our decision tree model are shown below.

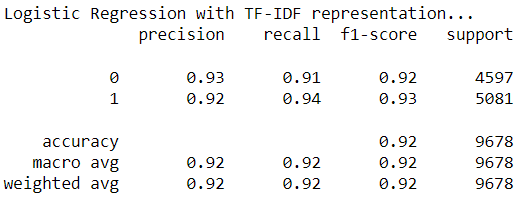




# **Logistic Regression**

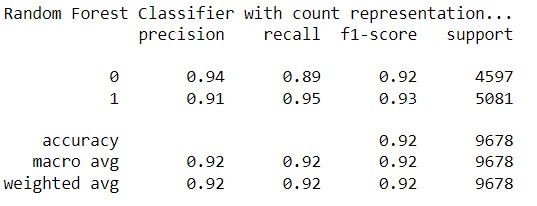
We used sklearn.linear\_model.LogisticRegression for our Logistic Regression model, and we set maximum iterations to 1000. This model determines the probability of an event occurring. This model works best on binary classification, such as the 0 and 1 in the label attribute. This model typically looks at independent variables to predict a dependent variable. In our case, the label was the dependent variable, and all other attributes used would be the independent variables. The results from this model are shown below.

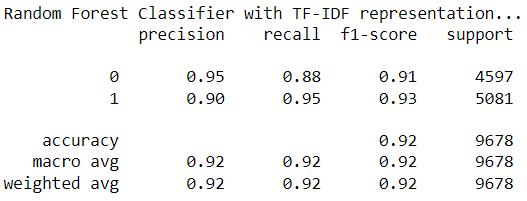




# **Random Forest Classifier**

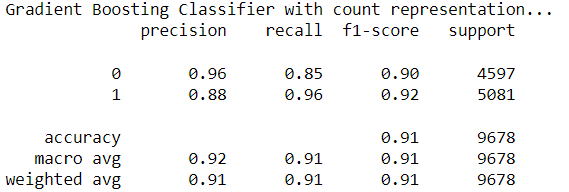
We used the sklearn.ensemble.RandomForestClassifier function for this model. This model essentially uses multiple decision tree models to classify the data, which helps to prevent overfitting and improve the model's accuracy. The accuracy is improved since the group of decision trees is more potent than any individual decision tree. The results are then grouped together, providing a more accurate model than if only decision trees were used. The results from this model are shown below.

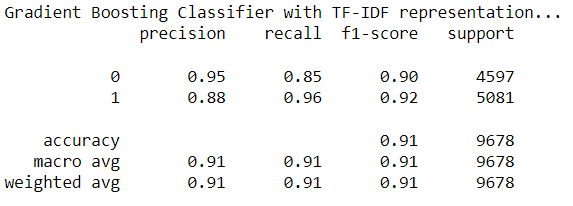




# **Gradient Boosting Classifier**

Gradient Boosting Classifiers is a class of machine learning techniques that combines several weak learning models to produce a powerful predicting model. It is the process of improving the strength of a weak hypothesis or learning algorithm by a series of minor adjustments. In other words, the GradientBoostingClasifier algorithm's fundamental principle is to develop models successively while attempting to minimize the mistakes of the prior model. This attempts to locate the inaccuracy left over from the previous model by using flaws from earlier iterations. Gradient Boosting repeatedly makes use of decision trees, and due to its success in classifying complicated datasets, they are gaining popularity. For the Gradient Boosting model in this project, we used sklearn's GradientBoostingClasifier function.

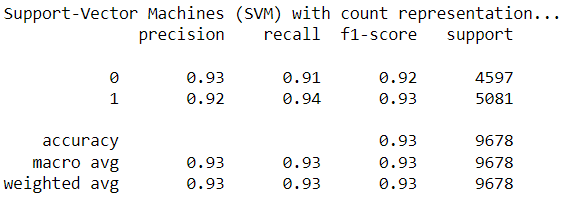


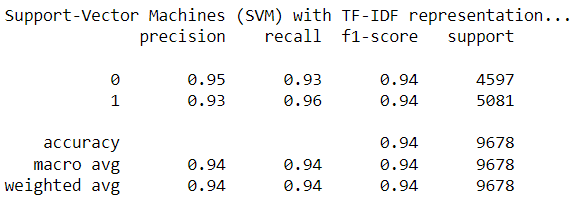


There are nearly no differences in all scores for Gradient Boosting Classifier using CountVectorizer and TfidfVectorizer. For the fake and real instances, the F1 score was 0.90 and 0.92, respectively, for both models.

# **Support-Vector Machines (SVM)**

A hyperplane is constructed in the supervised machine learning technique known as SVM in order to separate and classify data. Typically, the distance between each support vector on either side of the hyperplane must be maximized to choose the best hyperplane. In other words, the decision boundary between the feature categories will be more precise the farther apart each vector is from the hyperplane. For our project, we used sklearn’s SVC and fitted two SVMs to the vectorizers CountVectorizer and TfidfVectorizer. Specific parameters are needed for an SVM; in this dataset, we use 1.0 for C, “linear” for the kernel, degree of 3, and “auto” gamma.

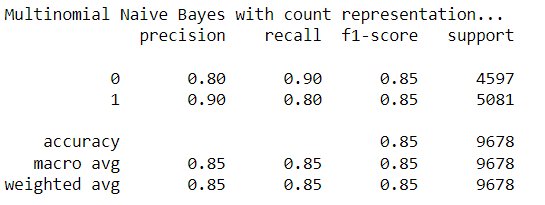


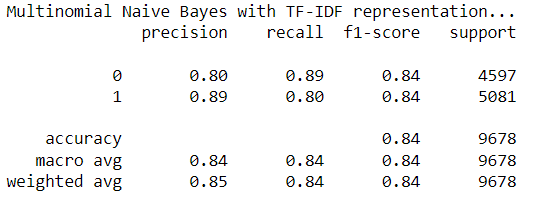


There is no significant difference between the score when using Count or TF-IDF Vectorizer as a preprocessing algorithm. The F1 score for fake news with CountVectorizer is 0.92, while with TfidfVectorizer, the score is 0.94. Similarly, the F1 score for real news of CountVectorizer is slightly lower, 0.93 with CountVectorizer and 0.94 with TfidfVectorizer.

# **Multinomial Naive Bayes**

We used sklearn.naive\_bayes.MultinomialNB for this model. This model uses the Bayes theorem in order to classify the label. More specifically, it calculates the probability of outcomes for each instance and then chooses the outcome with the highest probability. The Multinomial Naive Bayes model focuses on the frequency of words in a dictionary. The results from this model are shown below.





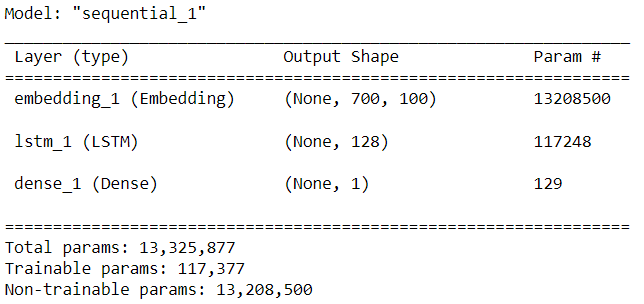
# **Neural Network with LSTM**

For the Neural Network model, we used a unique type of recurrent neural network called Long Short Term Memory networks (LSTMs). When training regular recurrent neural networks, the long-term dependency problem or the vanishing gradient problem may arise, and LSTMs can resolve these issues.

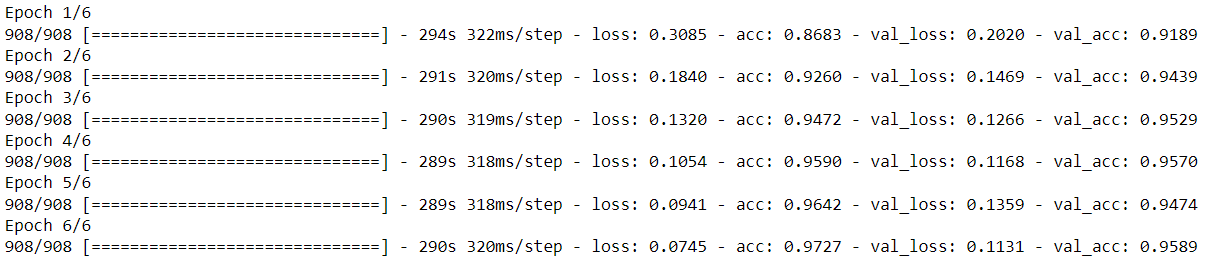
In this dataset, we used the TensorFlow, sklearn, and gensim libraries for the neural network. The Neural Network model uses interconnected nodes through different layers.

We need to use a method to display document vocabulary to train the neural network, and we chose word embedding. It can identify a word's position in a document, its semantic and syntactic similarities, its relationship to other words, etc. Keras's "Embedding Layer" implementation would produce word embeddings (vectors). We will load these vectors into the embedding layer and render the layer untrainable because we did so with gensim's word2vec. We made a matrix that maps vectors to word indexes, i.e., embedding layer's weights. The embedding layer receives a numerical word token and sends the appropriate vector to the inner layer. When a word is unknown, it transmits a vector of zeros to the following layer, where it will be tokenized as 0. The length of each news item served as the Embedding Layer's input length for our project. Overall, each raw text is converted into a fixed-size matrix during pre-processing. The LSTM unit is then used to train the model using the training data that has been processed.

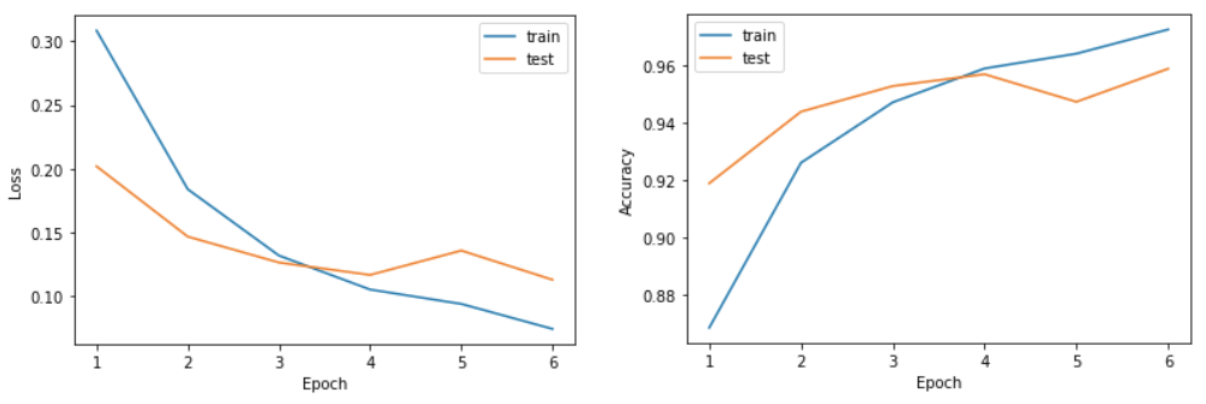
Since our project is a binary classification problem, we decided to use sigmoid as the output layer activation and binary\_crossentropy as the loss. With the Dense Layers feature of the Keras Sequential Model, we have connections to each hidden node in the subsequent layer. In this model, we used a different pre-training model, Word2Vec, one of the most well-liked methods for learning word embeddings using external neural networks.



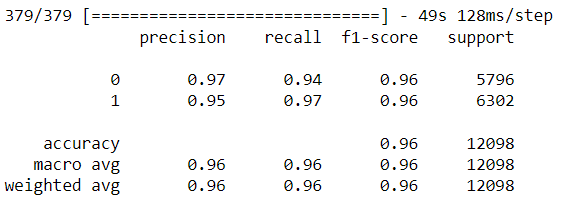
After running our model, we obtain the history of six epochs of the model, where the loss was in the range of 0.10 and 0.30, and the accuracy was in the range of 0.87 and 0.96, which shows that the epochs had decent performance. The epoch with the highest performance was epoch 6, and the epoch with the lowest performance was epoch 1.



In addition, the two graphs below show that the difference between the training and testing sets is not significant, which can confirm the decent performance of this neural network model.



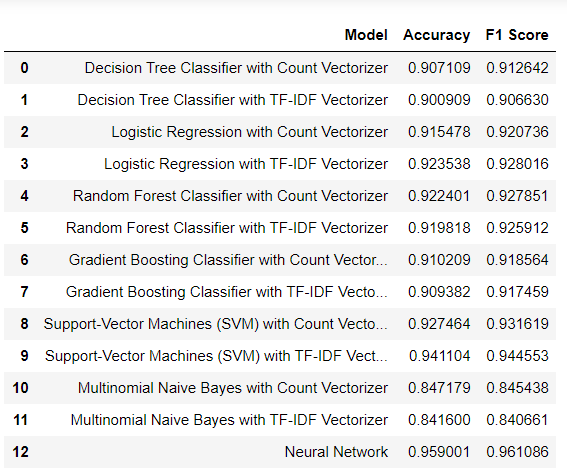
Finally, after training the Neural Network model with LSTM, we obtained 0.95 for fake news and 0.96 for real news F1 score. The averages were 0.95.



# 

# **Results**

# Our results found that the Neural Network model had the highest accuracy and F1 score, followed by the SVM model. The worst performing model was the Multinomial Naive Bayes model. In terms of overfitting, the Gradient Boosting Classifier and Neural Network models performed well. Most models had a difference of nearly 0.1, indicating that overfitting was detected in our model. We also found that the 80/20 split had better results than the 70/30 split. Most models performed exceptionally well, and almost all had accuracy and F1 scores of over 90%.



# 

# **Conclusion**

# In conclusion, the Neural Network model performed the best in terms of generalization performance, accuracy, and F1 score. In detail, the difference between the F1 score of the Neural Network model with training and testing score is unnoticeable, and the accuracy and F1 scores are 95.90% and 96.10%, respectively. The worst performing model was Multinomial Naive Bayes. The accuracy and F1 score of this model with Count Vectorizer was approximately 85% and with TF-IDF Vectorizer it was 84%. Additionally, this model performed poorly for overfitting, since the F1 score for the training of the CountVectorizer model was 95% and 85% for the testing. Similarly, for the TF-IDF Vectorizer of this model was 96% for the training and 84% for the testing. This study offers a method for encoding messages and examines how the general presence of words affects the distinction between authentic and fraudulent texts. In most of our algorithms, we got excellent F1 outcomes and accuracy. Most models performed well overall; however, overfitting appears in most of our models (except SVM, Gradient Boosting Classifier, and Neural Network).

# Hence, in the future, we will try to fine-tune the hyperparameters since, in this project, we ran out of time to do so. We intend to enhance our models’ performance by using hyperparameters or model tuning. Even though the accuracy and F1 scores are decent, the difference between the F1 score of testing and training data is large, and we have to handle that for better machine learning models. We can also enhance the model's training by utilizing various word embeddings, such as GloVe (Global Vectors). We can utilize a variety of text encoding formats that can be trained using these techniques to get a better model. Pre-trained models like Transformer and BERT can parallelize by switching out recursion for the attention method, so we can also try those. They can readily be adjusted in subsequent tasks because they need fewer processing resources. In order to ensure that nothing was overlooked, we would like to test our algorithm on new data in the future and redo the pre-processing and exploratory data analysis.