Mohammad Anas
Himangshu Raj Bhantana

IDS 703 Final Project

Introduction

The practice of classifying or labeling documents with categories based on their content is known as document classification. Document classification is critical for properly managing large volumes of unstructured text data in a variety of applications. Because we live in a digital age, the quantity of information available is endless, whether we are talking about a person or an organization. Document classification is a valuable tool that may decrease the cost and time spent searching for the relevant information within text and filtering it out to make decision.

Motivation

In our case, we chose Fake News detection as our NLP task as it has been all over the world for a few years now. Millions of people can quickly access social media and a news can quickly capture the attention of consumers. Major news companies have stated that not all the information supplied on social media is accurate, so we reasoned that by employing document classification, we would be able to handle the issue of whether a certain news is real or fake based on the frequency and placement of words.

Data

The data set chosen for this analysis was taken from Kaggle and contained information on various news articles and can be found here. It contained information on the text of news article, the title of the article, author and a binary label indicating whether the article was reliable (0) or unreliable (1).

General Data Preprocessing

Before data was used for analysis, some preprocessing steps were performed. All the text in the data were converted to lowercase alphabets. All the numeric or special characters such as "@ or $" were removed from the data as well. Furthermore, the stop words were removed from data. Stop words are words that carry no meaning such as "the, a, or and". We used Porter Stemmer to reduce all the words back to their base form for example words like drinking became drink. The NLTK package in python was utilized to perform these steps.

Train and Test Data

The whole data contained 18,285 rows and the first 15,000 rows were used as train data. This data was further split into train test splits for hyperparameter tuning. All our preprocessing functions were just fit to this data and the model was also optimized on this data.

The rest of the 3,285 rows were used as test data. These rows were treated as completely unseen data meaning the model and preprocessing steps finalized from train data were used on this. No separate preprocessing fits were used for this model. This test data was used to measure all performance evaluation metrics.

Naïve Bayes

We use Naïve Bayes as generative model for our classification problem. There are number of reasons for this. Given that the dimension of our data increase exponentially when preprocessing our text, we used it for its simplicity and ability to train quickly and efficiently. However, one limitation of naïve bayes is that it assumes that the features are independent of each other. Before we proceed onto how we use it to classify fake news, lets discuss how it generally works. The picture below is the simplest representation of how it works.

$$\underset{\text{Posterior Probability}}{P(c \mid x)} = \frac{\overset{\text{Likelihood}}{P(x \mid c)}\,\overset{\text{Class Prior Probability}}{P(c)}}{\underset{\text{Predictor Prior Probability}}{P(x)}}$$

$$P(c \mid \mathrm{X}) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

where c = class label,
x= feature words

When the train data was given to the model, model calculated the probabilities $P(X_i|C)$, P(C) and P(Xi) from it and uses these probabilities to generate the posterior probability. When given the test data, it calculates the probability of each class label given the features of each sentence (unigrams and bigrams in our case) and then assigns the class label for which we have the maximum probability.

Data Preprocessing for Naïve Bayes

Before applying Naïve Bayes, we need to convert our text data into words. We use the Bag-of-Words model for that. The columns or features names are selected using sklearn’s count vectorizer function. We create tokens from our sentences such that each token is either a single word (unigram) or a combination of two successive words (bigram). Once, we have all possible values of unigrams and bigrams within our text, we selection the 10,000 most common ones as features. Each vector represents the number of these features occurring in a sentence. The number of unique words in the all the train data were found to be 104,354.

Laplace Smoothing

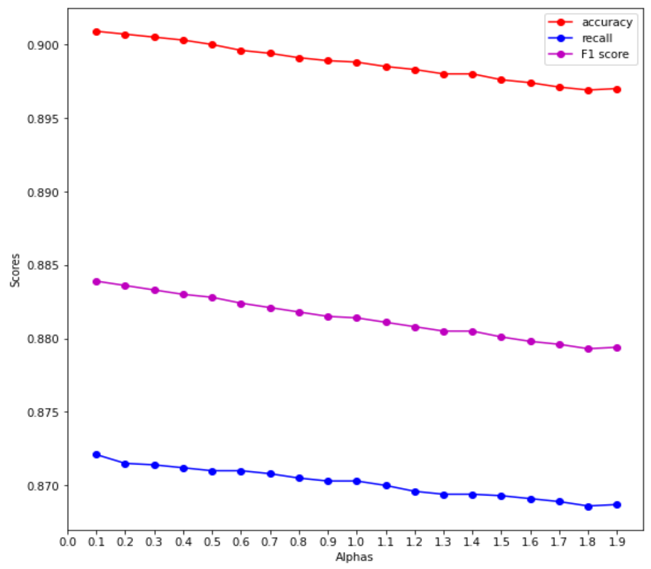
To ensure that our model can handle the ‘out of data’ words and does not assign them a probability of zero, we used Laplace Smoothing. This needed to be done because during the testing phase, the model would assign unknown words a probability of zero and when this is multiplied with other probabilities, the whole probability becomes zero, invalidating the model. The formulae for Laplace Smoothing is provided below.

$$p_{i,\ \alpha\text{-smoothed}} = \frac{x_i + \alpha}{N + \alpha d},$$

where alpha = smoothing constant,
d = total number of dimensions
N = number of comments with a i class label
X_i = the number of documents with i class label and that particular word
P_i= probability of word given i class label $P(X|Y_i)$

Training Model

K-fold cross validation was used to split our train data into 5 train-test splits. The model was then trained on each train split and evaluated on the test split. During the training procedure, the value of Laplace Smoothing constant was selected as a hyperparameter. The hyperparameter tuning was then performed along with the k-fold cross validation to select the optimal smoothing constant. The plot shows average of the evaluation metrics across all train-test splits for each alpha.



Based on the results from our results from k-fold cross validation, the alpha was set to 0.1 as it led to the highest accuracy, recall and f1-score. Moreover, it does not lead to over fitting. Using this hyperparameter, the model was then fit to the whole train data set.

Neural Network

There are a few limitations in our approach that we have taken to classify fake news. One of the limitations is that we have only been considering the frequency of each word occurring in a sentence using the Bag-of-words model. The sequential order has not been given importance. Moreover, the naïve bayes is also unable to handle this as it assumes that occurrence of each word in a sentence is independent of the other words, which is clearly not true.

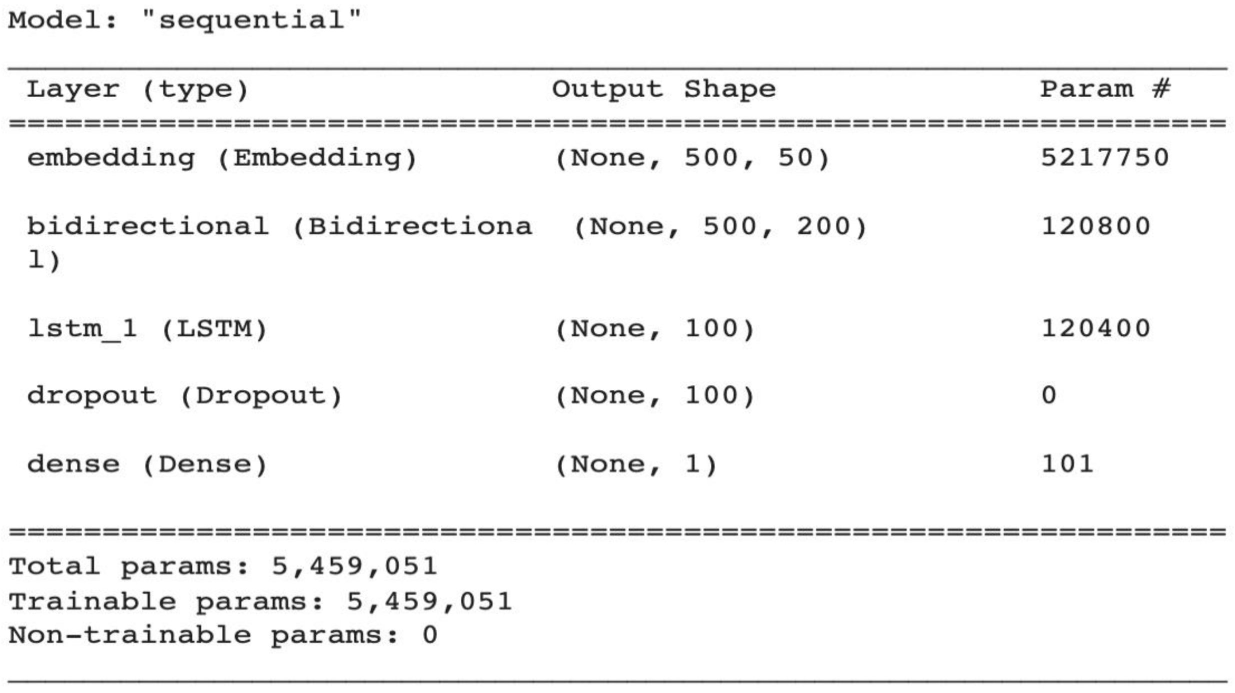
To counter this, we create another model using deep neural networks. We use Recurrent Neural Networks as they do consider the sequential order of words occurring within a sentence. However, one problem with normal RNNs is that they are unable to retain memory or mathematically speaking, they face the issue of vanishing gradients (gradient becoming extremely small). This leads to minimal update of weights during back propagation. Therefore, we turn towards LSTMs which rely on the concept of cell state that carries only relevant information when moving from one word to the next. However, a unidirectional LSTM also fails to consider the full context of each word in a sentence. The reason is that the meaning of each word is not only dependent on the previous words but also on the future words. Therefore, we need another LSTM that take one word after another in the reverse direction. This thought process led us to include a Bidirectional LSTM layer in our model, that happens to be one of the most important layers in our neural network design. Moving forward, we explain the full design and implementation of our neural network including the preprocessing steps.

Data Preprocessing for RNN

We convert our words into vectors using the tokenizer function in the Keras library. This allows us to assign a unique integer to each word that occur throughout a corpus. The function takes our total vocabulary length as an arguememt, that is the number of unique words in our corpus to do this efficiently. There 104,354 unique words in all our documents combined. Moreover, we also add an argument to this tokenizer function to enhance its ability to handle unseen words by assigning them an <OOV> tag. This might not be helpful in training but might help us when working on testing data. The function creates an <OOV> tag assigns it a new integer and adds it to the vocab size. Now that we have each word in an integer form, the next issue we face is the sentences of varying lengths. We use pad sequences in Keras to handle this. We covert each sentence to length of 500 integers (each integer representing a word), by either adding zeros to a sentence whose length was less than 500 or truncating sentences whose length was greater than 500 words. Now our data is ready to be fed to our neural network.

Design and Layers

The design of neural networks comprises of 5 layers: embedding, BILSTM, LSTM, Dropout and Dense.



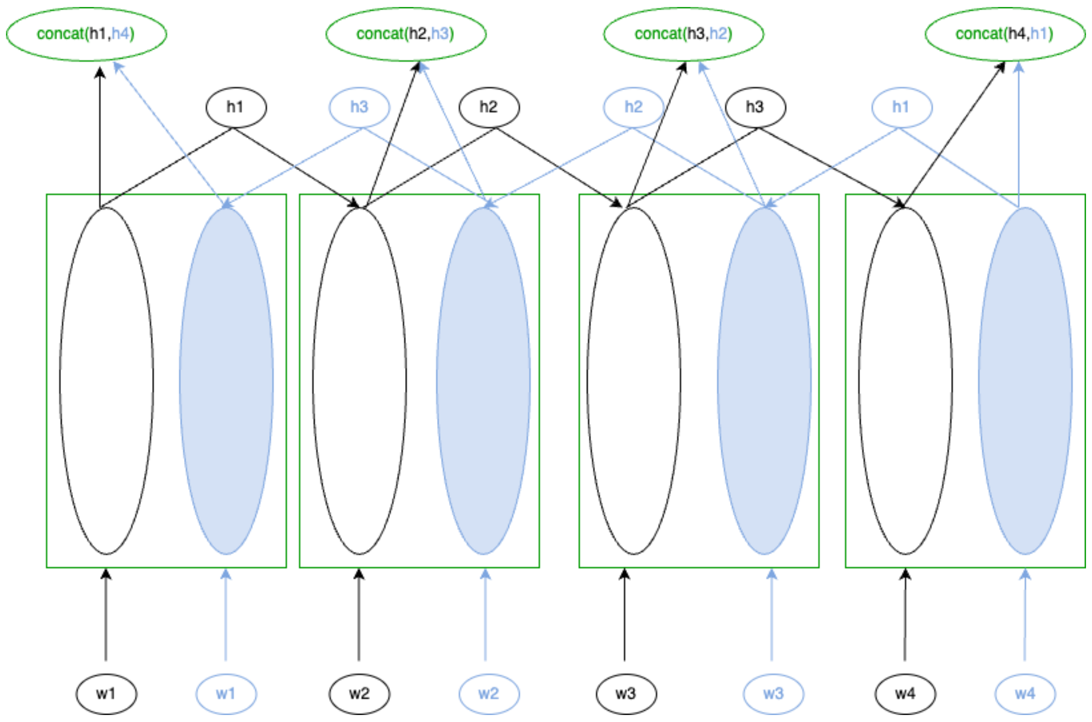
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 500, 50)           5217750

 bidirectional (Bidirectiona  (None, 500, 200)         120800
 l)

 lstm_1 (LSTM)               (None, 100)               120400

 dropout (Dropout)           (None, 100)               0

 dense (Dense)               (None, 1)                 101

=================================================================
Total params: 5,459,051
Trainable params: 5,459,051
Non-trainable params: 0
_________________________________________________________________
```

Embedding Layer

The embedding layer takes in the vectors that we get from tokenizer. We explicitly tell it our vocabulary size and the input length of sentences. For the vocabulary size here, we add one to our initial vocabulary size to take into account the <OOV> tag. Moreover, the dimension of each output word will be 50. After passing through the embedding the layer the dimensions of a vector corresponding to each word in our corpus has been reduced from 104355 to 50. Intuitively, the embedding layers adds context to are vectors in such a way that vectors corresponding to similar words will be pointing in the same direction in R_n (real number space) or their dot product will be higher. The output from our embedding layer will be a vector of the shape (500, 50), as 500 is length of each input sentence and 50 is the dimension of each word vector.

Bidirectional LSTM Layer

We use a bidirectional LSTM layer to ensure that full context of words. This layer takes in an input vector of shape (500,50) as outputted by the embedding layer. Moreover, we pass a return sequence argument to this class in Python to get every hidden state corresponding to each word. Each Bidirectional LSTM unit will further have 100 units within it and given that each word will pass through an LSTM unit twice, the hidden state corresponding to each word will be of dimension 200. This is because each unit within each LSTM unit will return an output of one dimension. Let's assume for a moment that we have 4 words in a sentence. Our BiLSTM layer will look like this.

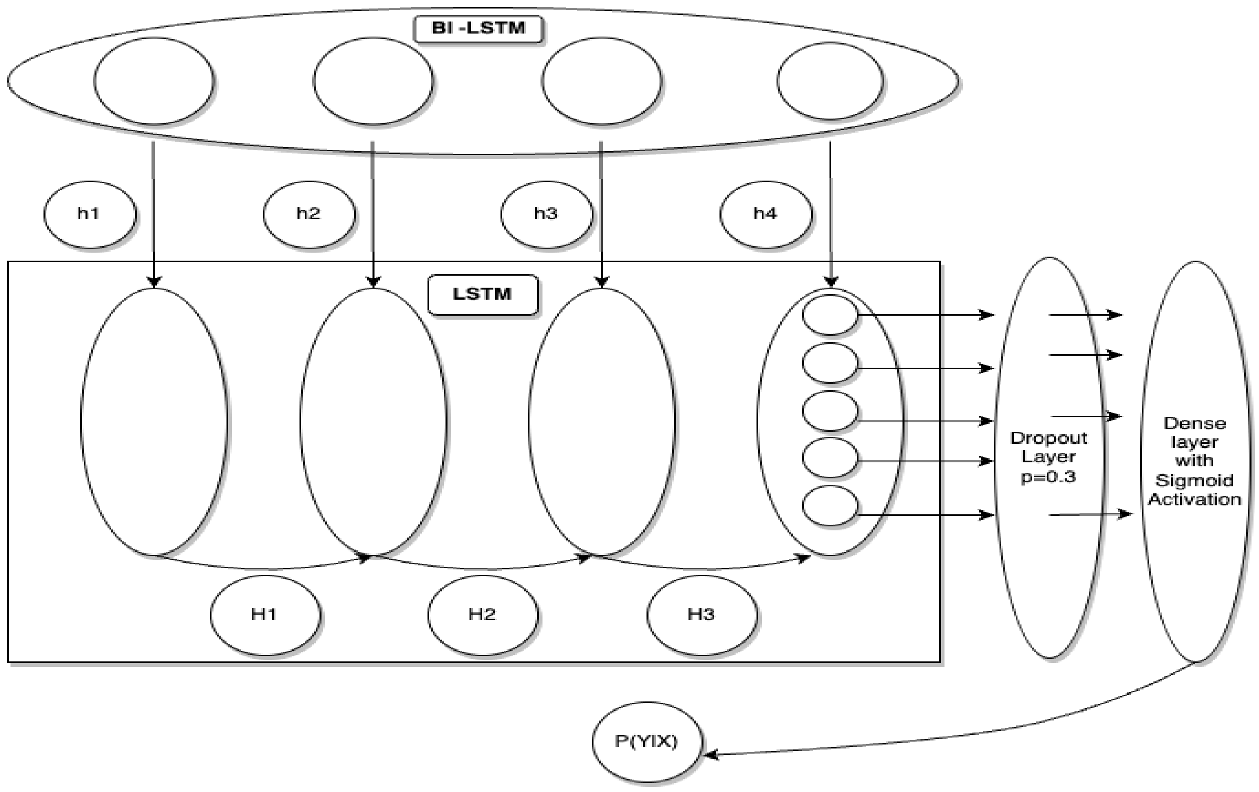


The green color in the diagram above corresponds to the Bidirectional LSTM as whole. Black indicates the LSTM going in normal given order and blue represents the LSTM that takes words in reverse order. The hidden states corresponding to each word from each LSTM will be concatenated (merge_mode = concat set as default in keras) together to form a hidden state outputted by the BiLSTM layer. The activation function for each LSTM unit was tanh (default in keras). Now each hidden state has all the relevant information for each word based on the previous and the succeeding words in a sentence. The output shape is (500, 200), meaning 500 hidden states and each of dimension 200.

LSTM, Dropout and Dense Layer

Each hidden state from the Bi-LSTM layer is then passed to an LSTM layer as input to be processed in sequential order. This LSTM takes an input of (500,200) and returns just one vector of 100 dimensions. The hidden states are not returned this time and simply passed to the next LSTM, only the hidden state of the last LSTM unit is returned. Each LSTM unit has 100 units within.

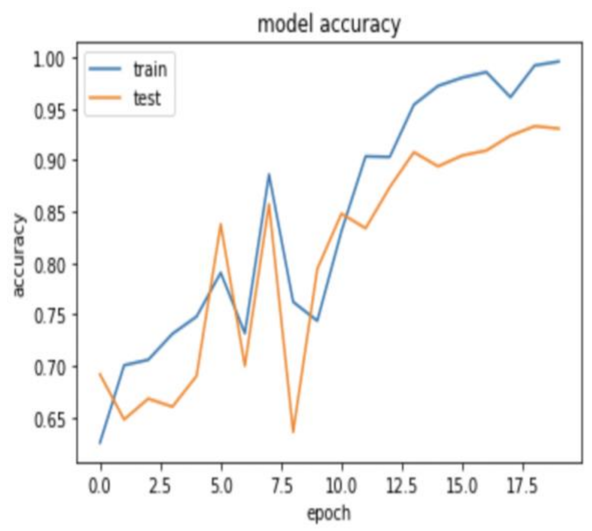
After the LSTM layer, a dropout layer is added, dropping output from 30 percent of the neurons in the last LSTM unit. This is done to prevent overfitting in our model. It should be noted that this does not change the output shape of the LSTM layer. The output remains of dimension 100, but 30 of these numbers in this vector will be zero. This is because the weights assigned with 30 percent of the neurons in the last LSTM unit were fixed to zero.

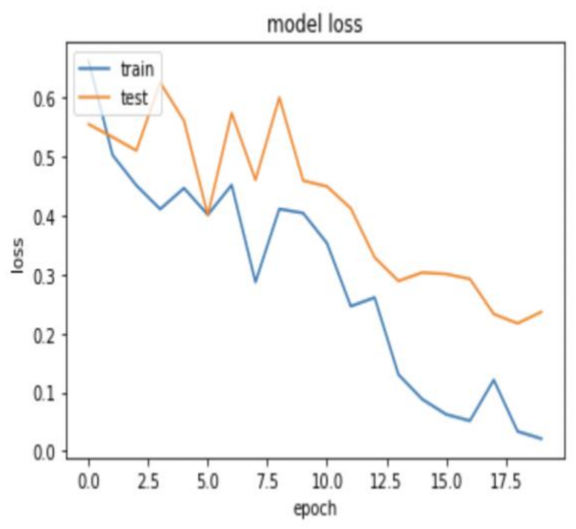


Finally, the output from the dropout layer of dimension 100 is passed into a dense layer. A dense layer is deeply connected layer within which each neuron takes input from all the neurons in the previous layers. In our case, the dense layer has only one neuron unit as we want our output to be simply a single number. Given that our problem is a binary classification problem, we chose **sigmoid function** as an activation function. This returns us a probability of a particular document being fake, which is then used to classify fake news. If the returned probability was above 0.5, the document was classified as fake news otherwise reliable.

Model Training

Our output label had two classes, so we used binary cross entropy as loss function. The learning rate was set to 0.001 (default in keras) and 20 epochs were used. This means that each input was run back and forth through the model 20 times. We initially used stochastic gradient descent to train our model. However, we soon realized that a lot more than 20 epochs would be needed to train our model with ‘sgd’ as optimizer. Therefore, we turned to adaptive learning rate optimizer like “adam”. This method takes large learning rate the start and smaller learning rate when close to the minimum of loss function. The data was split into train test split as the results of training the model on these splits are shown below.





Synthetic Data

For generating synthetic data, we used Naïve Bayes generative probabilistic model. We create a bag of words model for our whole data with the most common unigrams and bigrams as features. We fit the sklearn's Multinomial Naïve Bayes on this bag of words. The model calculates the following probabilities from the data.

$$\begin{aligned} P(X|Y_i) \\ P(Y) \\ \textbf{where} \\ Y_i = \text{class label i} \\ X = \text{words} \end{aligned}$$

We will use these probabilities along with the NumPy package's random functions to create our synthetic data. We extract the P(X|Y) from our model, which indicates probabilities of the features given a class label. These probabilities are extracted in form of an array of size 10,000 as it is the number of features that we have in our data. Each number in the array tells us the probability of each word occurring in each class label. We get two such arrays, one for each class label. We also extract the probability of each class label P(Y) in our data to get an idea of distribution of our response variable.

In total, we created 2000, data entries. We use NumPy's binomial random function along with our P(Y) to generate an array of 2000 size with 2 possible values of class labels. This array follows a similar binomial distribution or has approximately the same proportion of each class label as our actual data set.

Now that we have generated our labels, we need to create sentences in accordance with these labels. Here we use Numpy's multinomial random function to create a multinomial distribution of words in each sentence. The previously extracted P(X|Y) array is passed into the NumPy's multinomial random function to create a vector indicating the number of times each word occurs in a sentence. However, these words or token are restricted to the 10,000 features used in our bag of words model. We also note that the length of sentences in our actual data is the different and needed to ensure that it remains that way in our synthetic data. To tackle this, we create an array indicating the length of each sentence in our actual data. Using NumPy's random choice function we extract a random number from this array and choose it as a length of the sentence to be generated. We do this to choose the length of every sentence created.

Given that both Naïve Bayes and Bag of Words models are unable to take order of words into account, we are unable to generate a sequence of words for our sentences. Therefore, once we have created English sentences from the vectors indicating frequency of words in each sentence, we shuffle the words in our sentence to create a final document.

Testing our Models

Within our data, we have a balanced distribution for class labels. Therefore, our focus for evaluation would be accuracy. However, we will also look at recall, precision, and f1-score.

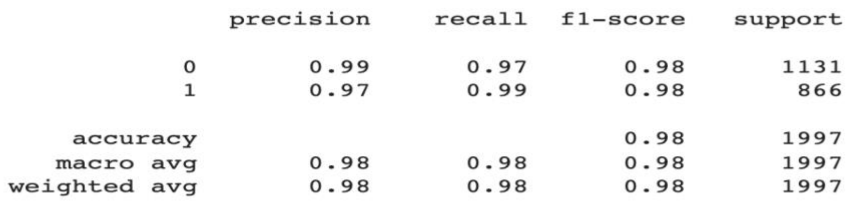
Testing our Models on Synthetic Data

To test our models, we treat the synthetic data as completely unseen data. This means that no new Bag of words or tokenizers were used for preprocessing the unseen data. Instead, the Bag of words and tokenizer fitted onto the training data were used to create a new bag of words (for naïve bayes) and one hot encoded vectors (for RNN) for the test data.

Naïve Bayes

It was no surprise to us when we saw an accuracy of 98%, using Naïve Bayes on synthetic data. The reason being that we had trained our generative Naïve Bayes on the whole data set and the Naïve Bayes model used for classification was trained on a huge proportion of the actual dataset. If we would have used the Naïve Bayes trained on the portion of data to generate synthetic data, would have achieved an accuracy of 100%, given that we were using the model with exact same parameters (probabilities) to generate and classify data.

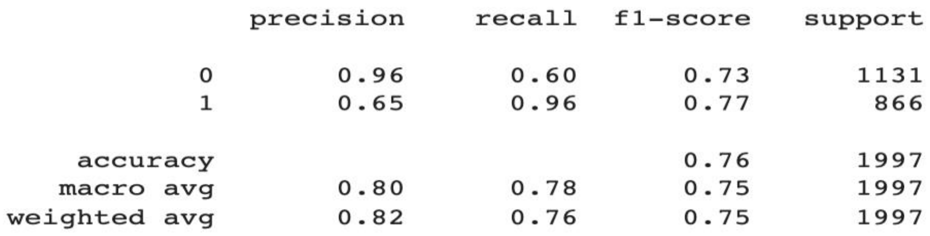
We achieved a recall of 99% on fake news, meaning that of documents that were fake news our model was able to classify 99% percent of them correctly. Of all documents categorized as fake news by our model, 97% percent of them were actually fake. We were able to achieve a recall of 97% and precision of 99% on the reliable news. The f1-score, a harmonic mean of precision and recall on both classes is 98%.



```
              precision    recall  f1-score   support

           0       0.99      0.97      0.98      1131
           1       0.97      0.99      0.98       866

    accuracy                           0.98      1997
   macro avg       0.98      0.98      0.98      1997
weighted avg       0.98      0.98      0.98      1997
```

Recurrent Neural Network

As for our discriminative model, our accuracy was 76%. The most probable reason for low accuracy was the lack of order in our synthetic data. Our RNN heavily relies on taking inputs in sequence. Not only does it consider the previous words but also the future words. Synthetic data was generated from Naïve Bayes, considering only frequency of words in each sentence. Hence, our RNN did not perform as well on synthetic data.



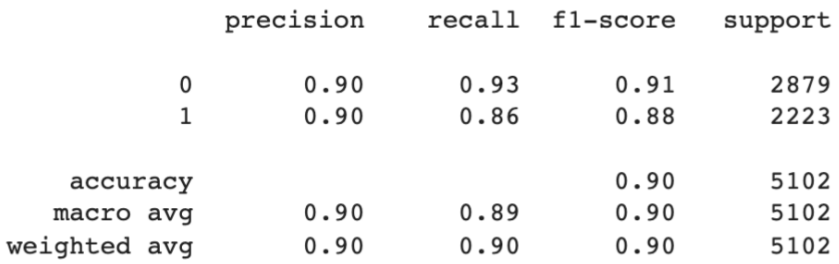
```
              precision    recall  f1-score   support

           0       0.96      0.60      0.73      1131
           1       0.65      0.96      0.77       866

    accuracy                           0.76      1997
   macro avg       0.80      0.78      0.75      1997
weighted avg       0.82      0.76      0.75      1997
```

Testing our Models on Legally Acquired Data

Now we test our models on an actual unseen data acquired legally. This data is the test data set aside in the start of the analysis and contains approximately 3,285 rows and has been untouched. We do some preprocessing steps on this like converting the text string to a lower case. We then remove stop words and stem words to their base form. All the special and numeric characters were removed.

Naïve Bayes

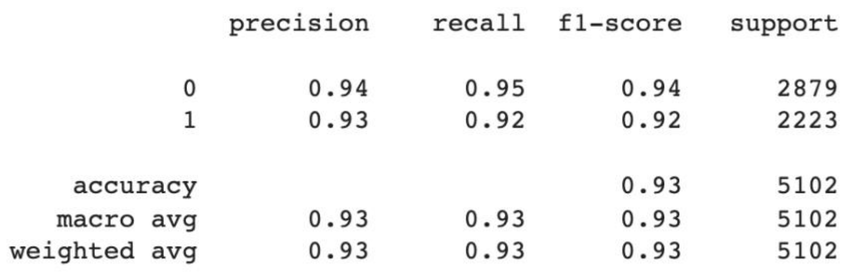
The count vectorizer fitted onto the train dataset was used to transform the text in train data to a bag-of-words. Our trained naïve bayes was applied to this bag of words and categorize sentences. This is where our smoothing technique helps. It assigns the unknown word a very low probability when calculating class label probabilities (P(Y|X). Hence, our model remains intact. Our model achieved an accuracy of 90% and did a better job at detecting unreliable news detecting 93% percent of them correctly. However, it classifies 86% of fake news correctly. The classification report looks as follows.



```
              precision    recall  f1-score   support

           0       0.90      0.93      0.91      2879
           1       0.90      0.86      0.88      2223

    accuracy                           0.90      5102
   macro avg       0.90      0.89      0.90      5102
weighted avg       0.90      0.90      0.90      5102
```

Recurrent Neural Network

We see that the RNN was able to outperform our Naïve Bayes in all evaluation metrics. A possible reason is that it does not assume independence of features assumption, intuitively meaning that it takes order of words into account. We were able to achieve an accuracy of 94%. The model does a brilliant job at classifying reliable news correctly achieving a recall of 95% on reliable news. It correctly classifies 92% of fake news correctly. The f1-score achieved was 94% on reliable news and 92% on fake news.



```
              precision    recall  f1-score   support

           0       0.94      0.95      0.94      2879
           1       0.93      0.92      0.92      2223

    accuracy                           0.93      5102
   macro avg       0.93      0.93      0.93      5102
weighted avg       0.93      0.93      0.93      5102
```

Pros and Cons of Models

To conclude, the major difference we see between models is that Naïve Bayes assumes independence of features while the RNN takes care of the order, handling context better. This the reason we see that it performs better on actual sentences. Given that synthetic data has no order to it and the sentences don't make sense intuitively, RNN is not able to decipher order as it has not seen it before. Generally, we see that RNN is a better model as it does well on classifying labels correctly. Moreover, it does better job at classifying the proportion of fake news and proportion of reliable news better compared to the Naïve Bayes. However, it should be noted that RNNs are harder to trains and require a lot of computational power. Moreover, BiLSTM and LSTMs do not entirely resolve the issue of losing memory or relevant information. They seem good for a document of 100 words but are not the best for sentences with 1000 words. This would require new state of the art technology like "Attention Mechanisms". RNNs also require a significant amount of data to train well. Naïve Bayes on the other hand is a simple approach that requires little data to train and hardly takes much time to do so. It performs exceptionally well if the features independence assumption is satisfied. Lastly, it does a great job at multiclass predictions as compared to other classification models.

Notes and Appendix

1. The code inform of jupyter notebooks will be attached on the homework zip file. Each of these will be named by a self-explanatory label.
2. Details of data can also be found on a team member's github repository. The repository includes all the data and code to perform this analysis. Please click here for that.