CS 4701 Project: Fake News Detector

Simar Kohli (sk2523), Esther Lee (esl86), Shefali Janorkar (skj28)

1. Introduction

1.1. The problem of fake news

As more and more information is used in our everyday lives, it becomes more important for us to be able to separate what is valuable truth from false distractions. This rising phenomenon of fake news has become even more widespread with the ongoing COVID-19 pandemic, and finding a way to identify and remove any sources of misinformation has become extremely important, especially as more people rely on technology to inform them of the state of the world.

1.2. What have Studies Shown?

An important thing to note is that fake news isn't simply false information; instead, the term is also used to classify pieces that rely heavily on opinion, non-news content, and lack of context to mislead and confuse readers. Because of this, verifying fake news isn't simply a process of fact-checking information. Instead, it requires a focus on actually learning the patterns and methodologies that legitimate news actually utilizes. This is why taking a machine learning approach to solving this problem has become an increasingly popular concept in recent years, and why it holds so much potential. (Molina, 2019)

1.3. Why use Machine Learning?

What potential, then, does machine learning have to solve this problem? Ultimately, the goal is to develop algorithms that can be trained with existing knowledge to produce hypotheses that predict future information. (Kotsiantis, 2007) Many algorithms have faced difficulty in finding specific indicators that are present in either real or fake news that correctly identify them because often, writing can be subjective and fiction can be presented as fact with very little effort. However, using a combination of different machine learning techniques, multiple approaches can be taken to find any such indicators or in general discover patterns that are not entirely obvious to human eyes.

1.4. Methods we Used

In order to accomplish the task of effectively classifying fake news, articles, etc., we used three machine learning supervised algorithms: N-grams, Naive Bayes, and Support Vector Machines. We trained the algorithms on a set of fake news articles, then hyper-parameterized
them based on their respective performances on the validation set, and finally compared their results, specifically accuracy, precision, and f1_score.

2. Pre-Processing & Data

2.1. Where did we get our data?

We found our datasets from a previous Cornell CS4740 Kaggle, and did a 80-20 split on our dataset, where 80% of the data was used to train the models and 20% of the data was used to test the models. We downloaded these datasets as text files — rather than as csv files — since it was easier for us to process them. However, we could not directly use the files, because they were unprocessed and were not optimized for creating natural language processing (NLP) models. So, we preprocessed the dataset.

2.2. Why preprocess?

Preprocessing is necessary when analyzing text and creating different NLP models, such as n-grams, naive-Bayes, and SVM models (which we have implemented for this project to detect fake vs. real news). Preprocessing eases the process of creating these models, and helps improve the accuracy of the model, as it removes all the trivial information. For example, when analyzing text, we do not want to include words that do not add any meaning to the overall content, such as 'a' or 'I' (stopwords). Keeping this trivial information can affect the overall accuracy of the model, since the model will be trained based on these datasets, which would mostly contain articles and stopwords. In this case, we would remove these words by implementing preprocessing functions. This would allow us to keep the important information and focus on the semantics of the text when developing accurate and robust NLP models.

2.3. Preprocessing methods and implementation

Our preprocessing method involved two main steps: tokenization and text cleaning.

We tokenized the text in our dataset using the nltk library's word_tokenize function, which breaks a text into different parts. These parts can be either words, punctuations, contractions, numbers, etc. All of the tokenized elements were stored in a Python array, which would be returned at the end of the function. Tokenization is an essential step in the preprocessing stage, as it allows us to remove stopwords, articles, and other elements that do not add meaning to the text in the later stages of preprocessing.

After tokenizing the text in the dataset, we "cleaned" the text by removing elements such as symbols (e.g. '(' and ')'). We accomplished this by scanning each element in the Python array and storing elements that were not stopwords, articles, etc. to a new Python array. This step was necessary because we wanted to focus on the semantics of the text, and including stopwords and other unnecessary words would prevent us from training and developing accurate models.

2.4. Experimenting with data

While implementing preprocessing functions, we have experimented with our data — the Kaggle dataset — and made sure that our preprocessing functions were working as intended. We used multiple print statements in between different preprocessing stages — tokenization and stopwords removal — to verify the outputs of the function.

As an example, this is a raw text extracted from our dataset that has not gone through preprocessing:

```
london (reuters) - britain is seeking to build on the  recent momentum  in the brexit divorce talks with the european uni
```

We notice that this is simply a string of text, which includes punctuations, symbols, space, etc. As our first step of preprocessing, we passed a list of text sources to a tokenization function, which would break up the text into different elements. If we pass the text above to our tokenization function, it would look like the following:

```
[['london', '(', 'reuters', ')', '-', 'britain', 'is', 'seeking', 'to', 'build', 'on', 'the', 'recent', 'momentum', 'in',
```

The text is now a Python array of different elements, including words, symbols, etc. As the final step of the preprocessing, we "cleaned" the text by removing symbols like "(," ")," or "-," etc. After cleaning the text, the array above would look like the following:

```
[['london', 'reuters', 'britain', 'is', 'seeking', 'to', 'build', 'on', 'the', 'recent', 'momentum', 'in', 'the', 'brexit
```

Finally, the symbols are removed from the array, and we see words such as "washington," or "trump," which are more meaningful.

As a side note, we have tested our functions in different articles/news across different datasets to make sure that the functions were working on different text data.

3. N-Grams

3.1. Theory

Our first area of experimentation was with N-grams, specifically, bigrams and unigrams. The motivation of the team in pursuing this course of learning model was due to previous experience with N-gram models, as well as the research done by Professor Cardie's lab here at Cornell (Ott, et. al.). The group showed that it was theoretically possible to utilize perplexity as a measure for determining how deceptive particular reviews were for products, locations, stays, etc. We theorized that due to the similarity in nature between fake/real news detection and fake/real review detection, that unigram/bigram models could sufficiently function as a form of machine-learning based detection.

N-grams function on simplistic probabilistic assumptions and mathematical relations. The principle is for a given corpus *P(w1, w2, w3, w4… wn)* we wish to estimate the probability of an occurrence of words. For a unigram, we are estimating the probability of the word *wi* appear within the corpus, or by extension, any corpus. For a bigram model, we are estimating the probability that the pair of words *wi* and *wi+1* occur together. By training two models (one for fake, and one for real), we're able to determine how likely it is for an input news article to be one or the other. Whichever model ascribes the highest probability (lowest perplexity) to the input, is the type of news it falls under.

Perplexity is a well known intrinsic metric for evaluation of learning models within NLP. In the context of text classification, using either perplexity or probability is typically considered acceptable. As such, we decided to proceed with analyzing the computed perplexity of the input under fake-trained bigrams and unigrams versus real-trained bigrams and unigrams.

3.2. Implementation

As stated before, the team focused on unigram and bigram models. One of the most significant reasons was staying away from higher-order n-grams that could result in too much contextual information (leading to overfitting to training data, and thus having a poor performance).

Bigram and unigrams were created using python dictionaries (words and pairs-of-words were mapped to their frequency). The bigram and unigram dictionaries were then modified inorder to handle unknown

```
def unigram_counts(lsts):
  map = {};
  for article in lsts:
    for word in article:
      if (word in map):
        map[word] = map[word] + 1
      else:
        map[word] = 1
  return map
```

```
[10] def bigram_counts(lsts):
       unigram = unigram_counts(lsts)
       bigram = {}
       for article in lsts:
         for idx, word in enumerate(article):
           if (idx != 0):
             f_word = article[idx-1]
             s_word = article[idx]
             key = str(f_word + " " + s_word)
             if (key not in bigram):
               bigram[key] = 1
             else:
               bigram[key] = bigram[key] + 1
       return unigram, bigram
```

Figure 1. Unigram, bigram code

words. This was accomplished by converting all mapped words that had a frequency/count below a threshold cut-off value (to be hyper parameterized upon later), into a mapped word called "<UNK>". "<UNK>" is referenced whenever a word is seen by the unigram that it has not seen in training. Similarly for bigrams, the unknown references were three cases: "<UNK> <UNK>" (neither words are seen), "<UNK> seen word" (only the second word is seen), and "seen word <UNK>" (only the first word is seen"). Unfortunately, one of the biggest issues with non-smoothed, MLE-based n-grams is the skewed probabilities we can see between words that may not have occurred together at all in training (and have 0 probabilities), and words that may happen together often (and have high probabilities). In order to counter this, we applied add-k smoothing. Quite literally, we add a proportionally changing value of k to all probabilities within the bigram and unigram. This makes 0 probabilities non-existent, and counters odd divide-by-zero errors, as well as doesn't make the model error out in the event it sees a pair of words (or word) it's never seen before.

Finally, we trained two versions of the unigram model (one for fake news, and one for real news), and two versions of the bigram model (one for fake news, and one for real news), on the given training data. We then conducted hyper-parameterization on the variables *cut_off* (representing the threshold value that would be the minimum number of counts/frequency required to not be converted to "<UNK>"), as well as *k_smoothing* (which represented the k value for the add-k smoothing). We hyper-parameterized values of *cut_off* at first from 0 to 10 on a step size of 1, and hyper-parameterized values of *k_smoothing* from 0 to 1 on a step size of 0.05. We then ended up reducing our range for *k_smoothing* to 0 to 0.05 on a step size of 0.01, based on the results we saw.

3.3. Results

The unigram and bigram models outperformed expectations significantly, with the unigram showing a better performance than the bigram. The unigram reached a maximum accuracy on the validation data of 95.23% with a *cut_off* value of 0 and a *k_smoothing* value of 0, and an accuracy of 95.41% on the test data. The bigram reached a maximum accuracy on the validation data of 91.38% with a *cut_off* value of 1, and a *k_smoothing* value of 0, and an accuracy of 91.38% on the test data.

Table 1. Results of validation and testing on the unigram and bigram models.

Models	Cut_off	K_smoothing	Validation Accuracy	Test Accuracy
Unigram	0	0.0	95.25%	**95.41%**
Bigram	1	0.0	91.23%	91.38%

4. Naive Bayes

4.1. Theory

Multinomial Naive Bayes is a common machine learning methodology used in text classification within the field of NLP. For instance, Poovaraghan, et. Al attempted to apply naive bayes in a similar way as we attempt to on a fake news test set. While their specific findings were somewhat vague, they clearly delineate how and why naive bayes could conceptually be a good model to apply. The learning model heavily relies on Bayesian probabilistic analysis, including the Bayesian assumption that the position of words do not matter within a sentence.

$$P(A|B) = \frac{P(A)*P(B|A)}{P(B)}$$

$$P(B|A) = Given\ a\ real\ or\ fake\ news\ article\ A,\ what's\ the\ probability\ of\ word(s)\ B\ showing$$

$$P(A) = What\ is\ the\ probability\ of\ document\ A\ showing$$

$$P(B) = What\ is\ the\ probability\ of\ the\ word(s)\ B\ showing$$

Our goal in the multinomial naive bayes is to generate a bag-of-words representation of the inputs (e.g. articles), and ascribe them a label regarding whether they are true or not. The naive bayes model then trains on the data, attempting to maximize P(A)*P(B|A) based on the label (either real or fake). The simplicity of naive bayes makes it a pretty efficient, and usually effective model for text classification, however the underlying assumption of the order of words not mattering can sometimes be one of the crucial factors for why naive bayes can sometimes underperform.

4.2. Implementation

The implementation of Naive Bayes within our experiment/project was relatively simple thanks to the utilization of sklearn.

We began by setting up a scikit pipeline using sklearn's *CountVectorizer()*, *TfidfTransformer()*, and *MultinomialNB()*. *CountVectorizer()* attempts to generate a sparse matrix of count frequencies of unique words within the input text (in this case, an article).

```
pipeline = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', MultinomialNB()),
])
```

Figure 2. Pipeline for naive bayes

TfidfTransformer() attempts to then take the sparse matrix of count frequencies, and remove words/counts that happen very infrequently. In this case, we apply add-1 smoothing to ensure that no divide-by-zero errors occur, and that unknown words still have a weight within predictions. Finally, *MultinomialNB()* is

the naive bayes classifier used on the actual training data itself. Note that since the sklearn naive bayes classifier has no effective parameters that can be hyper-parameterized on, we went with the default set-up (Laplacian smoothing).

4.3. Results

Due to the lack of hyper-parameterization (since the sklearn multinomial naive bayes is very limited in tunability), we simply trained the model on the training set and the validation set (due to no need for validation). We then applied the model on the test set and obtained an accuracy of 93.39%.

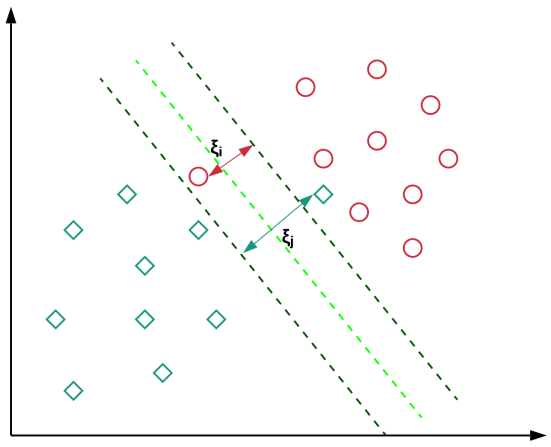
Table 2. Results of testing on the naive bayes model.

Model	**Applied Processing**	**Test Accuracy**
MultinomialNB	CountVectorizer, TfidfTransformer	**93.39%**

5. Support Vector Machines

5.1. Theory

Our final methodology for experimentation involved the support vector machine (SVM). The SVM is a supervised learning model that is capable of constructing a hyperplane on a high/infinite dimensional plane. This hyperplane essentially functions as a separation/barrier between different classifications (typically binary). Hussain et. al. show in their paper that an SVM is able to successfully determine fake vs real news media/information within Bangla news (Bangladeshi news), with very little training data.



Through the training of the SVM, it will attempt to utilize support vectors (data points that fall reasonably close to the soft-margin), in order to tweak the orientation and positioning of the actual hyperplane. Typically, there is a wide array of regularization and loss functions that can be used to change how the actual SVM operates. For the sake of simplicity and effective training time, we focused on using a linear, non-kernelized SVM.

Figure 3. Illustration of Linear SVM

Theoretically, manipulating these regularization and loss parameters can improve or degrade the model's performance depending on the type of data being inputted and the classification task. Most commonly, SVMs use a l2 norm with a squared_hinge loss. The regularization parameter C is used to - ideally - increase the strictness of the hyperplane (do we allow some data points to be within the hyperplane, or do we want to minimize this). A lower C value can produce softer margins, resulting in more data points being in the hyperplane, thus producing lower higher accuracies. However, training time is often sped up. A higher C value can produce harder margins, causing fewer data points to be allowed to be within the hyperplane boundary, producing higher training accuracies.

When using such SVMs, it's important to balance the C parameter such that we do not overfit to training data, but at the same time, maintain a degree of accuracy and specificity on data.

5.2. Implementation

The implementation of the SVM was straightforward, utilizing much of the similar sklearn packages and built-in functions.

A pipeline was generated using *CountVectorizer()*, *TfidfTransformer()*, *svm.LinearSVC()*. *CountVectorizer()* attempts to generate a sparse matrix of count frequencies of unique words within the input text (in this case, an article). *TfidfTransformer()* attempts to then take the sparse matrix of count frequencies, and remove words/counts that happen very infrequently. In this case, we apply add-1 smoothing to ensure that no divide-by-zero errors occur, and that unknown words still have a weight within predictions. Finally, *svm.LinearSVC()* is the actual linear SVM classifier used.

In this case, we apply a form of hyper-parameterization on the model using GridSearchCV. GridSearchCV takes in a given set of parameters, and applies a 5-cross-validation on the pipeline and

```
gridsearch = GridSearchCV(pipeline2, parameters, n_jobs =-1, cv = 5, verbose=4, scoring='accuracy')
gridsearch.fit(training, labels)
z = gridsearch.predict(nb_SVM_test)
```

Figure 3. GridSearchCV hyper-parameterization

the training data to output the best model with the best accuracy (and its corresponding parameters). GridSearchCV produces its own validation set (80-20 split) using the training data that is fed in, so our training data consists of both the validation and the training sets. The hyper-parameterization was done on the C-value (regularization parameter) to change how soft or hard the margin was on the range of 0.1, to 3.0. We then apply the optimal model on the test set generated, output our prediction, and compute our accuracy.

5.3. Results

GridSearchCV, after conducting 5 fittings each on 30 candidate values of C (resulting in a total of 150 fits), outputted a validation accuracy of 99.40%. The optimal value for the C parameter was a 2.2, and after applying this optimal model on the test set, we got an accuracy of 99.50%.

Table 3.

Models	**C-value**	**Validation Accuracy**	**Test Accuracy**
Linear SVM	2.2	99.40%	99.50%

6. Discussion & Conclusions

Ultimately, the results we have found are incredibly promising. Despite fake news being an incredibly nuanced issue that requires a lot of in-depth research to truly understand and combat, seeing these differing methods produce impressive results is a testament to the potential machine learning has in cases like this. Each specific method we tested had varying levels of accuracy, but all of them exceeded 90% accuracy and seemed to indicate a level of effectiveness. Overall, our SVM model produced the most accurate results with 99.5% accuracy — followed by unigram (95.41%), Naive-Bayes (93.39%), and bigram (91.38%) models.

Digging deeper into why exactly these models produced these results, our decision to not use higher-order n-grams was a good one; with a higher level of context, the model might have been overfit to the training data, so our already high accuracy score would not have meant much. A reason why our score may have been a good one could be that the n-gram estimation of the occurrence of a set of words could work well with common phrases and facts that are present in news articles that may be distorted or altered to become misleading information in fake news. What is interesting to note is that in our model, unigrams performed better than bigrams; almost as if more context actually had a detrimental effect on our results. Naive Bayes takes the route of considering unordered words, and this does appear to affect things; it has a slightly lower accuracy than n-grams and SVM. However, this is a very minor difference, so it appears that while word order can be important, it isn't necessarily the primary indication of text being fake news. Finally, SVM showed the most promising results of all three methods with its impressive accuracy over 99%. It isn't too clear why this method was the most effective, but perhaps its reliance on count frequencies was the key. Despite this high accuracy, SVM isn't without its drawbacks; the dataset we used was relatively small and allowed our classifier to more easily recognize patterns, but this may not have worked as well with a larger, more convoluted dataset.

Some intuition regarding performance drawbacks or successes involve the actual nuance of fake news. More often than not, fake news information tends to repeat the same content significantly (for

example, consistently saying that ‘Democrats are ruining America’, or ‘Trump has colluded with Putin directly’). As a result, the frequencies of certain words are expected to be significantly higher than other words. With bigrams still taking into account contextual clues (two words at a time), these frequency distributions (if they don’t occur often with each other), may lead the bigram-based learning model to think that the news is actually real. However, a unigram more astutely separates the occurrence of certain words with other words, and thus if it sees “Hillary” appear very often - regardless of whether it appears with “evil” or “good” - will look at the frequencies of other flag words as well to determine if the news article is fake.

It is likely that the SVM was able to perform so well due to its ability to generate a hyperplane on significantly higher dimensional fields rather than the multinomial naive bayes classifier’s reliance on the simplistic bayesian inference rule. Furthermore, it stands to reason that if a bigram (using the assumption that previous words influence the probability of a certain word showing up) was unable to perform well, that a classifier with similar probabilistic reasoning would also perform on a similar level. With the SVM’s ability to circumvent such assumptions, and ascribe the articles to higher complexity hyperplanes, it may have helped isolate certain keywords or phrases whose - on the specific hyper plane - presence or absence would delineate which side of the hyperplane the text fell.

As for the recommendations, there were some parts that we could have improved in our code. For example, for our preprocessing step, our function that cleans the text does not completely remove all the stopwords — it only removes all symbols and single-letter words that are not 'I' or 'a.' We could potentially improve this by removing more stopwords, such as 'the,' 'on,' or 'in' etc, since they do not add much meaning to the content of the article/text. Removing more stopwords could potentially increase the accuracy of the models, since these models would be focusing more on meaningful, "important" words, such as "britain" or "momentum" as shown in the example in 2.4.

As part of our future research, we could explore and research different NLP models to determine which models work best when building models that detect fake news. For example, this article has implemented different models, such as logistic regression, decision tree, and passive-aggressive classifier models, to detect fake news. For our project, we have implemented N-grams, Naive-Bayes, and SVM models to detect fake news, and determined that the SVM model produces the most accurate results out of the three. Since we were able to build an SVM model that has ~99.5% accuracy, we would be interested in researching other models that could produce even more accurate results.

7. References

Hussain, M. G., Hasan, M. R., Rahman, M., Protim, J., & Al Hasan, S. (2020, August). "Detection of bangla fake news using mnb and svm classifier." In *2020 International Conference on Computing, Electronics & Communications Engineering (iCCECE)* (pp. 81-85). IEEE.

Kotsiantis, S. B. (2007). "Supervised machine learning: A review of classification techniques." *Informatica*, vol. 31, pp. 249-268.

Kumari, K. (2021, July 19). *Detecting fake news with Natural Language Processing.* Analytics Vidhya. Retrieved December 10, 2021, from https://www.analyticsvidhya.com/blog/2021/07/detecting-fake-news-with-natural-language-processing/.

Molina, Maria D., et al. (2019). "'Fake News' Is Not Simply False Information: A Concept Explication and Taxonomy of Online Content." *American Behavioral Scientist*, vol. 65, no. 2, pp. 180–212.

Ott, M., Choi, Y., Cardie, C., & Hancock, J. T. (2011). "Finding deceptive opinion spam by any stretch of the imagination." *arXiv preprint arXiv:1107.4557*.

Poorvaraghan, R.J., Priya, M.V., Vamsi, P.V., Mewara, M., & Loganathan, S. (2019). "Fake news accuracy using naive bayes classifier." *IJRTE*:2277-3878

.