Variations of Naive Bayes and k-NN Classification for Gender-based Harassment Detection

Sarah Yoon, Nila Annadurai, Rachel Hong, Kofi Kwapong

January 24, 2019

1 Introduction

The purpose of this project is to develop a program that would be able to detect whether or not a sentence is considered gender based harassment. As a result, we built and analyzed several different types of classifiers for determining whether text is gendered or not. In order to do this, a training set of normal sentences and gender harassment sentences was used, collected from the 1965 revised list of Harvard sentences and Wikipedia Talk comment datasets from the Wikipedia Detox project and a collection of tweets used in a previous project and found on Data World. We labeled the formers as neutral/non-gendered and extracted/labeled gendered aggressive language from the latters comments.

We used three methods were used to test these sentences: parametric supervised learning through a Naive Bayes classifier and nonparametric supervised learning through K Nearest Neighbors classification. We implemented each algorithm using different word models in order to turn each sentence into a quantitative model. We also used Laplace Smoothing on all our variant models as well since after learning in lecture about smoothing improving results, we hypothesized that it would improve our efficiency.

2 Background and Related Work

Our basic Naive Bayes classifier is implemented as outlined by Fuchun Peng and Dale Schuurmans's paper, "Combining Naive Bayes and n-Gram Language Models for Text Classification" [2], and our bigram Naive Bayes classifier is implemented as the $n = 2$ case for (a simplified version of) the n-gram language model for text classification discussed later in the paper. We came up with our own set of features for determining whether or not a sentence is gendered, but based our implementation of our feature-based Naive Bayes classifier on Bharath Sriram, Dave Fuhry et al's work in "Short text classification in twitter to improve information filtering" [4].

As for our implementation of K-NN classification, we used the original "Nearest Neighbor Pattern Classification" paper by T. M. Cover and P. E. Hart [1] as reference. We cite the conference paper "Software Framework for Topic Modelling with Large Corpora" by Radim Rehurek and Petr Sojka [3] as our reference material for our usage of the Python Gensim Library for an alternative sentence vector model.

3 Problem Specification

The problem we are attempting to solve is the difficulty moderators of online public forums have when attempting to identify harassing, specifically gender-based harassment, in comments and discussions. We think this is particularly important in a day and age where AI algorithms are often biased in terms of gender or race. In order to do this, we construct multiple classifiers using various modifications of two primary algorithms - Naive Bayes and K Nearest Neighbors - and aimed to determine which methods are best in determining gender-based harassment.

More specifically for our Naive Bayes classifier, we wanted to obtain the accuracy, time efficiency, and memory efficiency of the following models for text: the classic Bag of Words model that we learned in lecture, a bigram model that stores occurrences of pairs of words (preserving order), and a feature-based model including four features of what we hypothesized would indicate a gendered sentence. Our four features were the number of offensive words (pulled from `http://sacraparental.com/2016/05/14/everyday-misogyny-122-subtly-sexist-words-women/`), the number of exclamation points which can indicate anger and passion, the number of commands which can indicate perceived power differences, and whether a sentence is considered hate speech, according to an open source hate speech API called Hate Sonar (`https://github.com/Hironsan/HateSonar`).

For the K Nearest Neighbors, we also planned to compare three separate models as well. First, we had a feature-based model which turned sentences into a two-dimensional vector of the sentence's level of hate speech, and the number of offensive words. Then we had a word vector model, which turned sentences into a vector v of n dimensions, where n was the number of words in our total dictionary, and $v_i = 1$ if the sentence contained word i. We also used a library vector model with the Python Gensim Library, which used several different NLP algorithms (outside the scope of this course) to convert text to a shorter vector that was intended to capture sentence meaning.

4 Approach

As inputs for our algorithms, we began by developing a list of labels that we classified as normal sentences or sentences that contain gender based harassment. We collected the normal sentences from the 1965 revised list of Harvard sentences, and we gathered the gendered sentences from the Wikipedia Talk comment data sets from the Wikipedia Detox project and a collection of tweets used in a previous project found on Data World. We parsed through this data and found the most relevant ones to our problem of identifying gender based harassment and then we collected our data into a text file by labelling normal sentences with a 0 and gendered sentences with a 1 preceding the sentence. Our training set had 300 gendered sentences and 300 non-gendered sentences, while our validation set had 142 gendered sentences and 216 non-gendered sentences.

4.1 Naive Bayes

We implemented our Naive Bayes classifier according to the algorithm below, in a very similar manner to the Naive Bayes classifier from problem set 5:

We utilize the self.counts structure, a 2-dimensional array, in order to store frequencies, calculated by the frequencies of sentences based on their features and a given rating. This allows

Algorithm 1 Outline of our Naive Bayes algorithm.

```
procedure NAIVE BAYES($b$)
    initialize dictionary mapping unique words in training_data to indices
    Initialize counts
    for $observation$ in training_data do
        $rating, sentence \leftarrow observation$
        for $word$ in sentence do
            Increment $counts[rating][word]$
        end for
    end for

    initialize F
    for $rating$ in [gendered, nongendered] do
        for $word$ in dictionary do
            $Pr(f_i|rating) \leftarrow \frac{k+counts[rating]}{counts[rating]+k*|counts[rating]|}$
            $word_i \leftarrow dictionary[word]$
            $F[rating][word_i] \leftarrow -log(Pr(f_i|rating))$
        end for
    end for
    for $observation$ in testing_data do
        $rating, sentence \leftarrow observation$
        $P_{gendered} \leftarrow Pr(gendered|sentence)$
        $P_{non-gendered} \leftarrow Pr(non - gendered|sentence)$
        $predicted = argmin(P_{gendered}, P_{non-gendered}$
    end for
end procedure
```

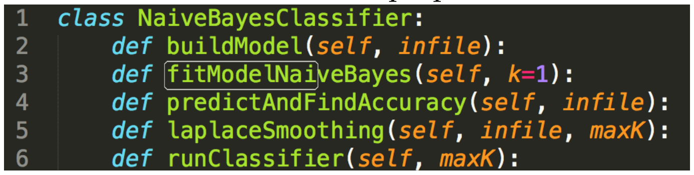
Algorithm 2 Outline of our Laplace Smoothing for Naive Bayes.

```
procedure LAPLACE SMOOTHING($b$)
    Initialize testing_k, results
    for $k$ in testing_k do
        FitModelNaiveBayes(k)
        Accuracy = predictAndFindAccuracy(testing_data)
        results.append(accuracy)
    end for
    return argmax(results)
end procedure
```

for efficient storage of multi-dimensional information within a single structure and thus gives us the ability to index into the data structure as necessary, particularly when computing probabilities to fill the model, self.F, with $Pr(feature|rating)$. Additionally, we utilize the dictionary for enumerating each feature. The dictionary structure allows us efficient, O(1) look-up and membership checking, due to python's underlying implementation of the dictionary as a hash table.

Finally, in the each of the 3 models, self.counts is instantiated differently, yet it relies on the same concept of storing frequencies. As referenced above, for the Bag of Words Classifier, the "features" are w_i (the index of a given word) which indicates whether a word is present in a sentence or not. In the Bigram model, the features are (w_i, w_j) pairs, similarly indicating whether the pair is present or not. Lastly, the Feature-based model uses the output of computed methods as features.

The algorithm for Laplace smoothing was essentially to iterate through all the possible k values, and add the k seen occurrences to our probability calculation. We then stored all accuracies across each k and outputted the k value which led to the highest accuracy. We iterated through k values in the set $[0, 1]$ since our frequencies in $self.counts$ were often pretty low, so we did not want the k value to dominate the proportion.



```python
class NaiveBayesClassifier:
    def buildModel(self, infile):
    def fitModelNaiveBayes(self, k=1):
    def predictAndFindAccuracy(self, infile):
    def laplaceSmoothing(self, infile, maxK):
    def runClassifier(self, maxK):
```

In terms of implementation in Python, for each different text model (bag of words, bigram, and features) we created child classes of a parent $NaiveBayesClassifier$ class, such that all the child classes were classifiers, but stored different data in the dictionary data structures. This allowed us to run each classifier in a very efficient manner in the main method, and also allowed our code to be rather structured.

4.2 KNN

For K Nearest Neighbors, we followed a very similar algorithm to the one presented in lecture:

In terms of data structures, the algorithm makes extensive use of the list, as it allows us to maintain groups of information and easily access individual elements within these groups. Specifically, we utilize lists to

1. Label our data. As each observation in training is pre-labeled, we can assign these observations into two lists to account for the distinction between observations labelled "gendered" and observations labelled "non-gendered". This allows us to later compute distances between testing observations and training observations, specifically in such a way that we have access to the not only the vector of the training observations, but also the given label associate with each training observation. The result is simple computation when determining the final label to assign to the new testing observations, as the result of the majority label of the closest "k" training observations.

2. Label our results. Once we determined the classification of testing observations, we made use of the list structure in order to compute the accuracy of our predicted classification when compared to the testing observations' true classification.

3. Store distances. We used lists to store distances between a new observation and training data, because this allowed us simple O(k) access to the first k elements (or the closest k distances)

Algorithm 3 Outline of our K Nearest Neighbors algorithm.

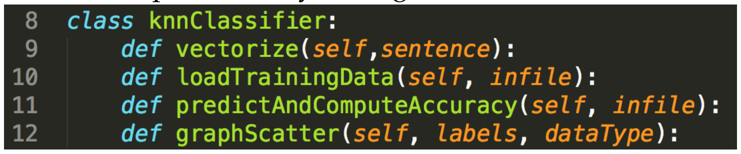
```
procedure KNN($training\_data, testing\_data, K$)
    initialize labels ( gendered  non-gendered)
    for $observation$ in training_data do
        $label, sentence \leftarrow observation$
        $sentence_{vec} \leftarrow vectorize(sentence)$
        assign $sentence_{vec}$ to $labels[label]$
    end for

    for $observation$ in testing_data do
        observation$_{vec} \leftarrow vectorize(observation)$
        $distances \leftarrow dist(observation_{vec}, vector)$ for $vector$ in training_data
        $closest_k \leftarrow$ closest K distances
        classify $observation$ according to most occurring label in $closest_k$
    end for
end procedure
```

of our computations. Thus, we could effectively classify these new observations within acceptable bounds as a result.

4. Represent vectors. Within the wordVector Classifier and the feature Classifier, lists also took on the role of the vector - in order to compute distances between observation, each classifier model had a different type of vector algorithm in order to represent an observation. Thus, lists allowed a universal method to represent varying dimensions and values of vectors, simplifying the code when it came time to compute distances on these different vector representations.

It is worth noting that the libraryVector classifier, while still making use of lists in the manner above, also utilizes the gensim Doc2Vec function to create a model structure. This structure, when given training data, attempts to incorporate semantic meaning into the vectorized sentences - the end result is a structure that permits us to capture the meaning of new observations during testing, and thus attempt to classify testing observations based on learned training observations.



```python
class knnClassifier:
    def vectorize(self,sentence):
    def loadTrainingData(self, infile):
    def predictAndComputeAccuracy(self, infile):
    def graphScatter(self, labels, dataType):
```

In terms of implementation in Python, for each different text model (word vector, gensim vector, and 2-feature vector) we created child classes of a parent *KnnClassifier* class, such that all the child classes were also classifiers, but had different *vectorize* methods in order to turn each sentence into a word vector. Just as with our Naive Bayes implementation, using polymorphism made our code cleaner and more readable, and also demonstrated similarities with the three models.

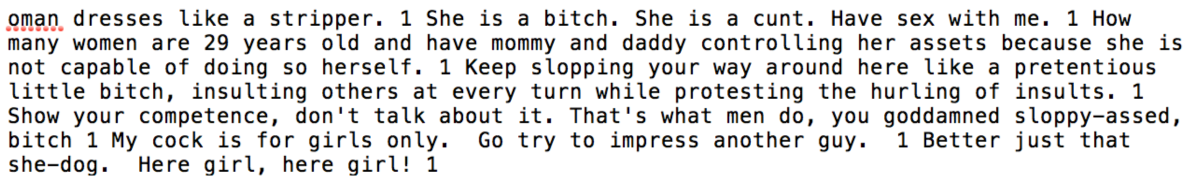
		Score
Naive Bayes Classifier (k = 1)		
Bag of Words (2334 words in dictionary)	Accuracy on Validation Set	0.8706
	Time (seconds)	0.1282
	Memory (bytes)	469648
	Accuracy with Laplace Smoothing	0.9109
	Good K for Laplace	0.164
Bigram (5642 words in dictionary)	Accuracy on Validation Set	0.4109
	Time (seconds)	0.6837
	Memory (bytes)	1664016
	Accuracy with Laplace Smoothing	0.4281
	Good K for Laplace	0.001
Feature Based	Accuracy on Validation Set	0.9425
	Time (seconds)	1040.2238
	Memory (bytes)	10384
	Accuracy with Laplace Smoothing	0.9425
	Good K for Laplace	0.1
K Nearest Neighbors (k = 2)		
Feature Based	Accuracy	0.9339
	Time on Training (seconds)	378.8695
	Time on Testing (seconds)	232.4585
	Training Memory (bytes)	65792
	Testing Memory (bytes)	95272
Word Vector	Accuracy	0.8390
	Time on Training (seconds)	0.0166
	Time on Testing (seconds)	54.0817
	Training Memory (bytes)	12476360
	Testing Memory (bytes)	19561848
Library Based Vector	Accuracy	0.5517
	Time on Training (seconds)	0.2253
	Time on Testing (seconds)	1.0609
	Training Memory (bytes)	3032368
	Testing Memory (bytes)	3068768

Table 1: Description of the results.

5 Experiments

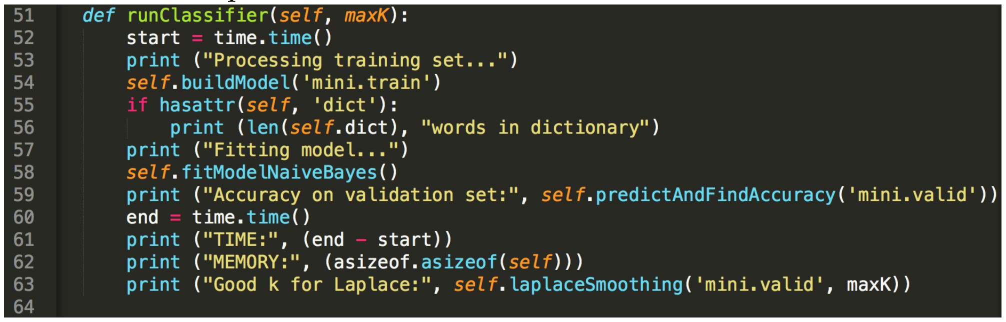
Below we have examples of testing data. These are just sentences that we have classified as gendered or normal, and we used these sentences to determine the accuracy of our various algorithms. To train our algorithms, we used 300 gendered sentences and 300 normal sentences. To test our data, we used 216 normal sentences and 142 gendered sentences.

```
The bank pressed for payment of the debt. 0 The theft of the pearl pin was kept secret. 0
Shake hands with this friendly girl. 0 The vast space stretched into the far distance. 0 A
rich farm is rare in this sandy waste. 0 His wide grin earned many friends. 0 Flax makes a
fine brand of paper. 0 Hurdle the pit with the aid of a long pole. 0 A strong bid may
scare your partner stiff. 0 Even a just cause needs power to win. 0 Peep under the tent
and see the clowns. 0 The leaf drifts along with a slow spin. 0 Cheap clothes are flashy
but don't last. 0 A thing of small note can cause despair. 0 Flood the mails with requests
for this book. 0 A thick coat of black paint covered all. 0 The pencil was cut to be sharp
at both ends. 0
```



```
oman dresses like a stripper. 1 She is a bitch. She is a cunt. Have sex with me. 1 How
many women are 29 years old and have mommy and daddy controlling her assets because she is
not capable of doing so herself. 1 Keep slopping your way around here like a pretentious
little bitch, insulting others at every turn while protesting the hurling of insults. 1
Show your competence, don't talk about it. That's what men do, you goddamned sloppy-assed,
bitch 1 My cock is for girls only.  Go try to impress another guy.  1 Better just that
she-dog.  Here girl, here girl! 1
```

Below is an example of the code we wrote in order to test our classifiers:



```python
def runClassifier(self, maxK):
    start = time.time()
    print ("Processing training set...")
    self.buildModel('mini.train')
    if hasattr(self, 'dict'):
        print (len(self.dict), "words in dictionary")
    print ("Fitting model...")
    self.fitModelNaiveBayes()
    print ("Accuracy on validation set:", self.predictAndFindAccuracy('mini.valid'))
    end = time.time()
    print ("TIME:", (end - start))
    print ("MEMORY:", (asizeof.asizeof(self)))
    print ("Good k for Laplace:", self.laplaceSmoothing('mini.valid', maxK))

```

5.1 Analysis and Critique

For unigram bag-of-words, our algorithm was expected to take linear time, taking $O(n)$ time in order to build the model, given a dataset of n distinct words, and $O(2n)$time in order to fit the model, since to build the model we simply iterate over all of the words in the file to populate `self.counts`, and to fit the model we iterate over words to calculate their probabilities for each rating (in this case there are only 2, gendered and not). As for space complexity, we expected unigram Naive Bayes to use size $O(2n)$ memory, because for each word we need to store the number of occurrences of the word in gendered sentences and the number of occurrences of the word in non-gendered sentences.

Bigram bag-of-words, however, was expected to take $O(n^2)$ time, because there would be $\leq \binom{n}{2}$ possible pairs of words rather than n words to iterate over, and $O(2n^2)$ memory for the same reason. Our feature-based Naive Bayes classifier, on the other hand, we expected to take $O(n)$ time, with this approximation omitting some constant n would be multiplied by in order to account for the fact that this algorithm iterates over sentences rather than words (and a constant number of features per sentence, each taking linear time to compute). However, due to the fact that one of our features, determining whether or not a given sentence is classified as hate speech by the Hate Sonar API, requires an API call, we expected this algorithm's runtime in practice to be much longer. We expected that the feature-based algorithm would have a smaller space complexity than bag-of-words and bigram NB because rather than having data structures to store counts for words, the amount of memory used would be proportional to the number of feature results for the sentences, which would be smaller than the number of words in the data set.

We expected Laplace smoothing to increase accuracy for each of these algorithms, with our value for k not changing the accuracy by much for $k > 0$. Having an n-gram classifier would generally increase accuracy for $n > 1$, so we expected that using a bigram classifier would increase accuracy, but for our limited dataset it was also possible that it would actually decrease accuracy because there would be many more pairs of words not seen in gendered sentences at all. We expected that ussing a feature-based classifier would likely increase accuracy because it takes into account factors having to do with tone and sentence meaning that indicate whether or not a sentence is gendered that are lost when evaluating sentences without taking word order into account.

For KNN, we expected all models to take a runtime of $O(nd + nlogn + kd)$, if n is the number of sentences in our training data. This is because for all n we compute the distance of the new observation, then we sort, which takes a runtime of $O(nlogn)$, then we calculate the first k distances, which is why we add kd. While the runtime is the same in complexity for all of the algorithms, we expect the time it takes for the feature based model and the Gensim library model to be much smaller. This is because the time it takes to calculate distances is much smaller as for the word vector, we are making a large vector of all the words so the time it takes to calculate distances is much longer. It is also possible that the feature based model would take a signifantly longer time beacause we are making API requests, which might add additional time. Therefore, we expect the Gensim library based model to take the least amount of time as it doesn't have an API requests and it the vector is significantly smaller than the word vector.

For memory, if n is the number of sentences and c is the average number of words per sentence, the memory complexity of feature based KNN model is $O(n)$ because we are storing one vector of four values per sentence. For a word vector based model, we are storing one vector of size n*c per sentence so the memory complexity is $O(c * n^2)$. For the library based model, the memory complexity is also $O(n)$ because we are storing a constant vector for each of our sentences. Therefore, we expect the word vector based to take much more memory than the other two models.

In terms of accuracy, we expected the feature based model to be the most accurate, because it is calculating the distances based on the exact features that we are looking for. Next we expected the Gensim Library vector based model to be most accurate, as it creates a vector based on meaning, which is ultimately what we are trying to determine: whether the meaning of a sentence is gendered and malicious. Finally we expected the laplace smoothing to increase the accuracy of each model.

For KNN feature-based model, we hypothesized that the higher the hate speech score and the higher the number of offensive words, the more likely the sentence would be gendered. Therefore, we conducted initial data analysis and graphed these two features on separate axes in Figure 1, and agreed that the graphs were pretty consistent with our assumptions that gendered sentences tended to have higher hate scores and more banned words, while almost all of the non-gendered sentences were at $(0, 0)$.

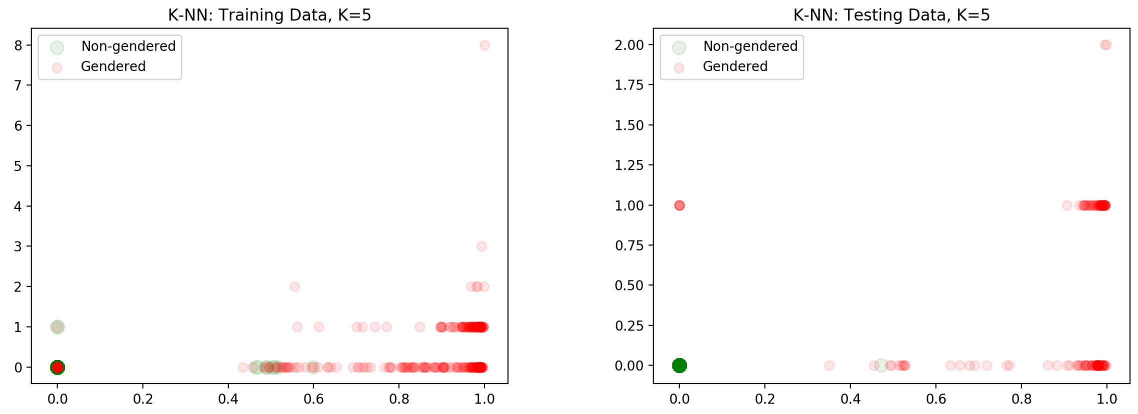
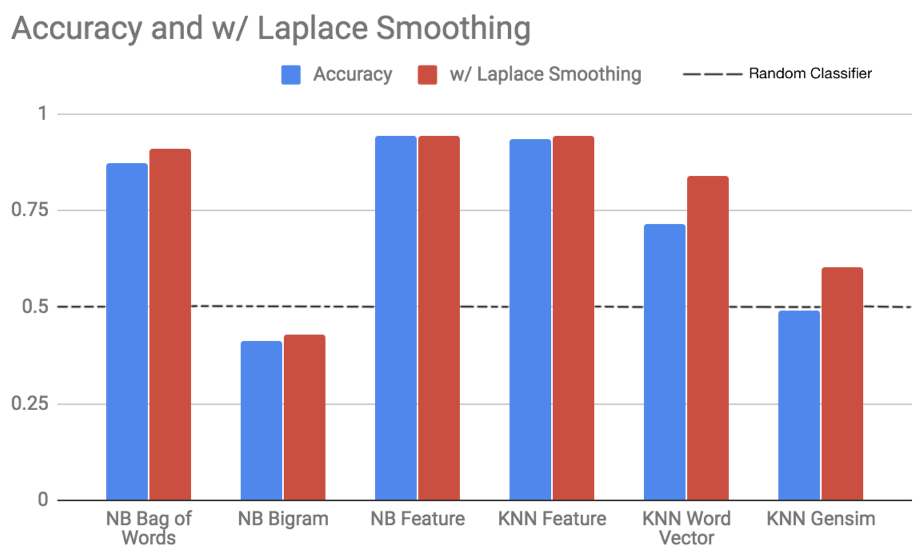


Figure 1: Categorizing sentences as feature vectors, X-axis: hate score, Y-axis: number of offensive words

5.2 Results & Evaluation

5.2.1 Accuracy

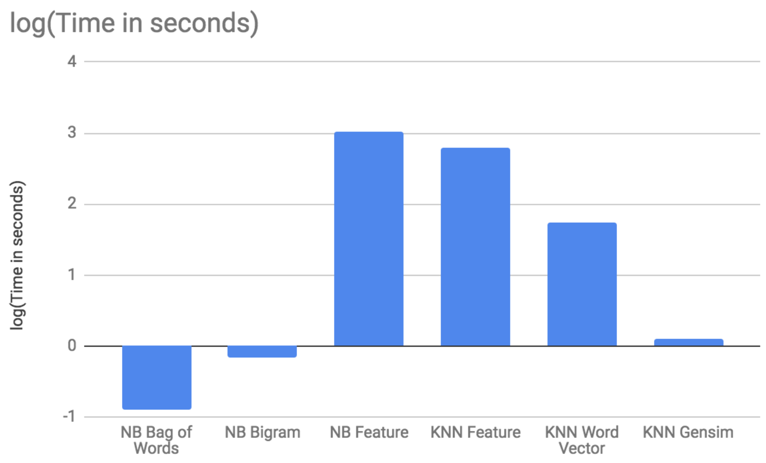


As expected, Laplace smoothing did indeed improve accuracy for most of the algorithms, verifying what we had learned in class. Though we predicted that our bigram classifier would increase accuracy relative to our unigram Naive Bayes classifier, as we suspected might happen bigram Naive Bayes actually performed far worse than unigram bag-of-words in terms of accuracy, due to the fact that our dataset was quite sparse, and the classifier had not seen the new pairs of words in our test data. This meant that our Naive Bayes bigram classifier actually performed *worse* compared to a classifier that allocated sentences randomly (since half of our training data set had gendered sentences). This was particularly shocking, but was also understandable since our algorithm did not increase the probability of a sentence being gendered or not, given a new bigram.

In addition, our feature-based Naive Bayes classifier did increase accuracy compared to bag-of-words, though our unigram bag-of-words algorithm already performed quite well.

Also as expected, the all KNN models benefited in accuracy with the Laplace smoothing. Also, the feature based KNN model was most accurate; however, the Gensim Library Vector Model wasn't as accurate as we expected, and the KNN word vector model was much more effective. This is most likely because much of the vocabulary in the gendered sentences were very similar so having a word based vector was very accurate at detection gender based harassment.

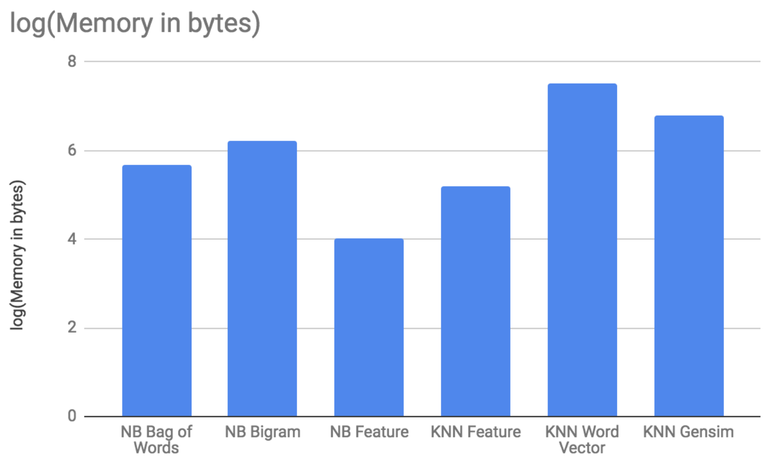
5.2.2 Runtime



As we expected, bag-of-words took less time than bigram NB classification, because of the difference in time complexity, and feature-based Naive Bayes took much longer due to the API calls necessary (although by a much larger margin than we might have predicted).

Also as expected, the KNN feature based model also took the most time as it was making API calls. Additionally, the Gensim model was significantly smaller than the word vector model, most likely because the vector was much smaller and thus the time it takes to calculate distance is much smaller.

5.2.3 Memory



The bigram NB classifier did indeed use more memory than our unigram classifier (due to having to store more items in our dictionary data structure), and as we predicted, the feature-based classifier used less than the other two because there were only 4 features to store the frequencies of, compared to over 2 thousand words, or 5 thousand pairs of words.

For the KNN models, we can see that as expected, the Word Vector model took up the most memory, as it is storing vectors with a size of the total unique number of words. The memory used by the Gensim Library model is smaller as it is storing vectors with a fixed size of 100, and finally as expected the feature based model is the smallest as for each sentence it only has to store a vector of size 2.

5.2.4 Optimal K Value

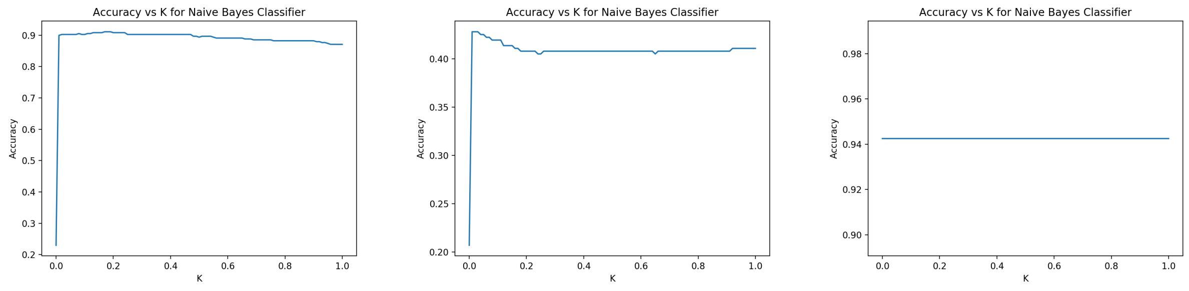


Figure 2: From left to right: Naive Bayes Classifier Bag of Words, Naive Bayes Classifier Bigram, Naive Bayes Classifier Features

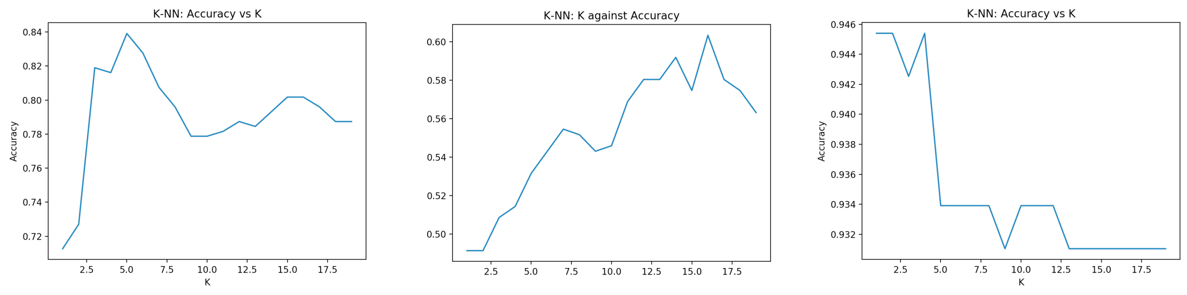


Figure 3: From left to right: KNN Classifier Word Vector, KNN Classifier Gen Sim Library, KNN Classifier Features

6 Discussion

As we can see from Table 1, we can see that the feature based models for both the Naive Bayes Classifier and the K Nearest Neighbors were the most accurate, and they were both comparably accurate with each other. This entails a trade off, however, as both of these algorithms require a significantly higher runtime than the other algorithms. This is most likely because while for the hate speech feature, we have to make an api request, which takes a significant amount of time. It also takes more time to characterize the features of each sentence, but having specific features and attributes that the classifier is looking for - specifically banned words, and commands - really helps the accuracy in determining gendered sentences.

The worst performing algorithm we tested was the Bigram Naive Bayes method. The primary reason why we believe it has such a low accuracy is because it stores pairs of words, and there were only 5000 pairs of words in our dictionary from our training data, and many new pairs of words in our testing data.

In terms of memory, the Bigram took more memory than the standard Bag of Words model because it recorded pairs of words instead of normal words. In addition, the feature based model took less memory because we were only storing features rather than storing every single word.

There are some improvements to this project we could implement in the future. For instance, due to the fact that calling the Sonar API significantly increased run time for our feature-based models both for KNN and Naive Bayes, we could build our own hate speech API as a seperate classifier, instead of using the Sonar API, which would increase efficiency.

If we had more time, we would also use cross validation for the optimal k value instead of determining the best k value for the given training set. We would also definitely include more comprehensive data for accurate results that could incorporate the nuance of gendered language. Right now many of our gendered sentences contain very similar vocabulary, which makes it us able to detect very crass gendered sentences in a successful manner. However, we could greatly expand our scope if we added many more sentences that were more diverse to detect more nuance in gendered sentences, as well as sentences with similar words but are categorized differently.

A System Description

1. The code repository, along with the training and test data, can be found at `https://github.com/hongrachel/cs182finalproject`

2. Packages to install:

   In order to run locally, in Python3, one will need to install the following packages: $nltk, hatesonar, pympler, matplotlib, smart\_open, gensym$

3. In command line

   run $python3$ $naivebayesclassifiers.py$ for Naive Bayes classifiers

   run $python3$ $knn.py$ for KNN classifiers

B Group Makeup

- Nila Annadurai, Sarah Yoon – finding, parsing and labeling datasets, interpreting the algorithms for accuracy and efficiency, creating the poster
- Rachel Hong, Kofi Kwapong – implementing and applying the algorithms to the data, finding and implementing improvements to these algorithms (e.g. Laplace smoothing and finding k), turning sentences into feature vectors

References

[1] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13:21–27, 1967.

[2] Fuchun Peng and Dale Schuurmans. Combining naive bayes and n-gram language models for text classification. *Sebastiani F. (eds) Advances in Information Retrieval. ECIR 2003. Lecture Notes in Computer Science*, 2633:335–350, 2003.

[3] Radim Rehurek and Petr Sojka. Software framework for topic modelling with large corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50. Masaryk University, Faculty of Informatics, 2010.

[4] Bharath Srirarm, Dave Fuhry, Engin Demir, Hakan Ferhatosmanoglu, and Murat Demirbas. Short text classification in twitter to improve information filtering. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pages 841–842. ACM, 2010.