Naive Bayes Document Classification

Dante Welch[1]

[1] *Grand Valley State University*

In machine learning, Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Bayes' theorem describes the probability of an event based on prior knowledge of conditions that may be related to the event. Using these ideas, we can perform sentiment analysis and classify documents as either positive or negative quite easily. In this paper, I will be classifying documents as either having positive or negative sentiment based on previously labeled documents.

I. INTRODUCTION

As discussed in lecture, Naive Bayes Classification uses probabilities to estimate whether or not a given document belongs to one class or another. The term 'naive' is used due to the fact that the algorithm treats each feature as an independent attribute. In this writeup, I will specifically be classifying IMDB movie reviews, but the algorithm is generalizable and can be extended to plenty of other domains such as sentiment of tweets or detecting spam. As human beings, we are able to simply use our intuition and prior understanding of the language we speak in order to classify a given document as either positive or negative. For example, the word 'stupendous' will likely occur in the positive documents much more often than in the negative documents. Therefore, if the classifier sees the word 'stupendous' in a document, it will likely classify it as positive. Obviously, this is a contrived example, but the actual algorithm essentially does the same thing.

II. PREPROCESSING

In its raw form, the dataset (like most) is not ready to be used with any type of machine learning algorithm and needs some preprocessing done to it. One of the main goals of preprocessing is to filter out useless data, and the most obvious way to do that is to remove stop words from the dataset. A stop word is a commonly used word (such as 'the', 'a', 'an', 'in') that is very unlikely to tell us anything about the sentiment of the sentence. In order to easily collect a list of English stop words, I used the popular stop words package from the Natural Language Toolkit. Once I had every stop word available, the first step was to tokenize a document by word and strip all punctuation/capitalization away. From there, I checked a document for the stop words to be removed. In addition, I decided to remove all words with a length of 1 in the document.a For each document in both the negative and positive directories, the following code was executed to preprocess the document:

```
def clean_doc(doc):
        tokens = doc.split()
        table = str.maketrans('', '', string.punctuation)
        tokens = [w.translate(table) for w in tokens]
        tokens = [word for word in tokens if word.isalpha()]
        stop_words = set(stopwords.words('english'))
        tokens = [w for w in tokens if not w in stop_words]
        tokens = [word for word in tokens if len(word) > 1]
        return tokens
```

In addition to cleaning the document, I also decided to further explore stemming as a means of creating a more effective classifier. After some searching around, I decided to use the package PorterStemmer, which implements an effective stemming algorithm backed by substantial research. Using the PorterStemmer package made it trivial to stem each document. For each word in a document, simply use the built in method, stem, and return the list of stemmed words. In order to stem each document, the following code was used:

```
def stem_doc(tokens):
        ps = PorterStemmer()
        stemmed_doc = []
        for word in tokens:
                stemmed_word = ps.stem(word)
                stemmed_doc.append(stemmed_word)
        return stemmed_doc
```

The raw dataset had roughly 138,000 unique words present, and after stemming, it dropped to roughly 89,000. This was not the only measure of implementing a more efficient algorithm, but it certainly helped substantially in regards to runtime.

III. BUILDING THE CLASSIFIER

In order to build the classifier, I stored each unique stemmed word in a file, vocan.txt, where each word was its own line in the file. Using vocab.txt, I was able to iterate over each word, and check each document, both positive and negative, and keep a count of how many times each word was used with respect to its document's sentiment. I stored each of these values in a dictionary, with the key-value pairs looking something like the following:

```
{
 ‘for ’: (986,1127), ‘movi’: (21797,27800),
 ‘get ’: (6430,7609), ‘respect ’: (490,364),
 ‘sure ’: (1409,1638), ‘lot ’: (2457,2267),
 ‘memor’: (484,212), ‘quot ’: (154,130),
 ‘list ’: (316,404), ‘gem’: (317,95)
 }
```

With the first value in the tuple being the number of times the word occurred in a positive review and the second value being the number of times the word occurred in a negative review. From here, it's simply a matter of looking at a document, cleaning it, and using its bag of words and Naive Bayes to make a prediction for its classification.

IV. EFFECTIVENESS OF THE CLASSIFIER

An interesting problem came about when I was attempting to classify documents in the test sets. Essentially, every probability was approaching zero and it was not able to make any predictions. After doing some research, I found out that Python was causing my floating point values to lose precision, and it was recognizing very small numbers as essentially zero. To combat this, I used the built in Decimal class in order to keep precision through my calculations.

After solving the problem of losing floating point precision, I was able to train my classifier on 25,000 training documents, split evenly between positive and negative reviews. With that, I was able to test my classifier and have a maximum accuracy of 83% on the test set, which was also comprised of 25,000 documents split evenly among positive and negative reviews. One way for me to possibly improve the accuracy of my classifier would be to have more training data, so I could have (in theory) used some of the testing data as training data. Having more training data would not cause overfitting because generally, Naive Bayes is unlikely to overfit due to its linear nature. The place where my model ran into the most trouble was on reviews that fell into roughly the middle 25% of reviews. By middle 25%, I mean reviews that may be either positive or negative, but not polarizing in either direction. That being said, the model rarely classified highly polar reviews incorrectly, which is the main reason why Naive Bayes is known as a solid algorithm for sentiment classification.

V. IMPLEMENTATION ISSUES

One unique problem I ran into while testing my classifier was with the stemming library I chose to use. After creating a corpus of words from every training document, when testing, there were words that when stemmed, did not exist in the vocabulary. Some of the words, one being 'backdraft', were complex enough to not be able to be stemmed properly by the algorithm. The way I went about fixing this was to assume that these words were rare enough to not impact the classification much and to simply skip over them if they were not present in the keys of the dictionary.