NLP Coursework

John Steward

1 Data Splitting

When reading in all of the .txt files from the dataset, I wanted to make sure my training set, evaluation set and testing set all had the same proportion of positive and negative reviews to keep them as fair as possible. In order to achieve this, I iterated through the folder of positive reviews and negative reviews separately, adding the first 1600 positive reviews into a positive review list, adding the next 200 into an evaluation list and the last 200 into a test list. I then did the same with the negative review folder. This resulted in having 3200 training datapoints, 400 evaluation datapoints, and 400 testing datapoints, each with a 50/50 split of positive and negative reviews. This gave me an 80/10/10 split between training, evaluation, and test sets which means I can train on plenty of data, while still having enough data for evaluation and testing to give meaningful results. Since I iterate through the positive and negative txt files only once each, there is no possibility of cross-contamination between datasets. I also store all of the tokens for each set in their own lists to calculate the frequency distribution for storing in a dictionary. I put all the positive training datapoints into a list, all of the negative training datapoints into a list, the evaluation datapoints into a list and all of the testing datapoints into a list. This way, I can guarantee that I will only have access to the correct subsets when training, evaluating and doing my final tests, without danger of cross-contamination skewing my results. Having each in their own lists also means I can easily test different tokenisation methods while still having access to all of the original data in an easily usable format. In order to vectorise each document later on, I keep a list of the documents as a whole, split up by their tokens, and I also keep a list of all the tokens in order to form my vocabulary to use for training and testing.

2 Feature Generation

2.1 Tokenisation and Cut-off

For my experiments, I used 3 different techniques for tokenisation: splitting by whitespace, stemming, and lemmatisation.
Splitting by whitespace is the simplest implementation, taking the piece of text and splitting each word by spaces, which means keeping all punctuation within words. This can be a flawed approach, since words that are the same, but have different punctuation before or after will be stored as different tokens in our vocabulary. For example, we could have the words 'Great', and '(Great' as separate tokens, when we can clearly see that they are the same word.
The next technique that was implemented was stemming. I implemented stemming from the NLTK library, after tokenising each word from the documents using NLTK word_tokenise to initially split the words. The stemming function takes each of these tokens and truncates them, reducing them to their base form, removing any tense context or plural context. For example, a stemming algorithm would take the words 'changing', 'changes' or 'changed', and would reduce them all to 'chang'. This eliminates the issue from splitting by whitespace, since it would generally reduce similar tokens to the same base word, meaning they will all be stored the same. This could still present its own issues, as it may not take homonyms into account, and may stem a word with one meaning, as the same word with a different meaning.
The final tokenisation method that was implemented was lemmatisation. This has a similar function to stemming, while still being slightly different. While stemming truncates a word into its base form, lemmatisation reduces words down to their base dictionary form, its 'lemma'. Using the same example as used for stemming, 'changing', 'changes', and 'changed' would all be reduced to their base word: 'change'. This takes a little longer to run than stemming, since stemming relies on strict rules that are

hard coded, whereas lemmatisation relies on a lookup table. When using stemming, the word 'better' would be reduced to 'bet' whereas, with lemmatisation, it would be reduced to 'good'. In cases such as this, stemming may lose the base meaning of the word, as bet could be interpreted in different ways, but lemmatisation keeps the base meaning of the word, however the issue of homonyms may still present itself here, and it uses the most likely meaning of a specific word, basing that probability on the volume of the word in the language. In my implementations, I found that lemmatisation was generally a better method than stemming, despite taking longer to run (See table 4).

Even when we group together tokens using stemming and lemmatisation, we may still end up with a vocabulary that is much too large to feasibly work with, especially when the size of the dataset increases. When we increase the length of our vocabulary, then we also increase the length of our vectorised documents, and the time it takes to process the IDF values for the vocabulary. This means that, as we add more tokens, computing time can dramatically increase. When we tokenise our documents and look at the frequency of each of the tokens, the words that are most common tend to be uninformative for our classification, as they occur many times in both of our classes. The most common words tend to be articles such as 'a' and 'the'. We want to remove these from our vocabulary so we do not compute these and we vastly reduce the dimensionality of our problem space. These words can be removed by implementing NLTK's stopwords function to remove a hard-coded list of words that are uninformative to any classification tasks. Despite using this, we can still end up with many uninformative words that occur much too often in our dataset, so we want to implement a cut-off value for the number of instances of the words.
We also end up having a very long tail of words that only occur maybe once or twice in the dataset, generally ones that are specific to one film, rather than the reviewer's sentiment towards the film. We also want to remove these for the same reason as we want to remove the most common words. When choosing our cutoff, we may want to find a balance between the optimal performance of our classifier and the speed of computation. For example, we may get a higher accuracy when using all of our initial tokens, but it may potentially take a lot longer than if we only use half. We may choose to use half if the accuracy is not much lower than when we use all of the tokens. In practice, the optimal cutoff varied based on the tokenisation and method of extracting phrases (See tables 1 and 3).

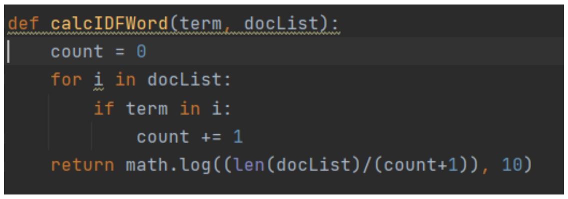
2.2 Information Extraction

As well as extracting individual words from our documents, we also want to be able to extract commonly used phrases as these often give more information than the individual words. For example, in a single sentence, we may have the word 'not' followed by 'good'. When we extract just the individual words, we have a positive and a negative word, so they cancel each other out, giving us a neutral sentiment for the phrase. When we look at the phrase 'not good' as a whole, we know that it has a negative sentiment, and so is more likely to appear as a whole phrase within a negative review, whereas the words 'not' and 'good' individually may appear more in both classes.
The first method that I implemented for extracting compositional phrases was using Parts-of-Speech Tagging (PoS) and constituency parsing. In order to get the correct PoS tag for each word, I cannot stem or lemmatise the words first, as that may cause them to lose their contextual meaning and therefore be given the incorrect tag. So, I pass the raw tokens into the NLTK pos_tag function to get the tags for each word, then I pass those tags into NLTK's RegexParser in order to build a constituency tree for each sentence. Using this tree, I extract all phrases that are tagged with 'NP', meaning they are noun phrases. To do this, I iterate through the sub-leaves of the constituency tree and add all of the words within each noun phrases to a string, then add the string to a dictionary, and to a list of all the phrases in each document in order to process the term frequency during vectorisation.
The other method I used to extract compositional phrases was storing frequently occurring n-grams. I found that, generally, using 3 word n-grams was the optimal solution for performance (See table 3). The method for doing this was to iterate through the list of tokens in each document, storing every 3-word string in a dictionary. For example, using the sentence "I walked into the room", I would extract "I walked into", "walked into the", and "into the room". I would then cut elements out in the same way as I did with the individual words, with the same frequency cut-offs. I then stem these and add them to the original dictionary. This is a much simpler approach than using PoS and constituency parsing, but may take longer, since I am iterating through every word and storing every 3-word phrase, and may often store irrelevant information. even if a specific phrase occurs enough times to be stored

in the vocabulary. However this can also be the case with PoS as our algorithm has not learnt which phrases contain a relevant sentiment, it is just learning the frequency of words and phrases within the documents.

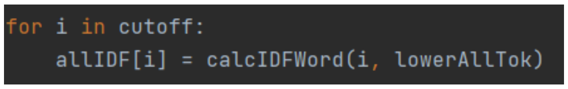
2.3 Normalisation

In order to get a more accurate interpretation of the sentiment from each term, we want to normalise them. We want terms that appear more often in our query document and in our vocabulary to be given more weight than those that occur less frequently. One way of doing this is with the TF-IDF algorithm. We first calculate the Inverse Document Frequency of each of the terms in our vocabulary by iterating through our vocabulary, passing each term and a list of all the training documents, which contains a list of each token in the document. In this function, we iterate through each document in the list, and, if our term is somewhere in the document, we increment a counter to find how many documents the term is found in. We then return $\log(\frac{number\_of\_documents}{counter+1})$ (see figure 1a). Since we are calculating the



```python
def calcIDFWord(term, docList):
    count = 0
    for i in docList:
        if term in i:
            count += 1
    return math.log((len(docList)/(count+1)), 10)
```

(a) IDF Calculation

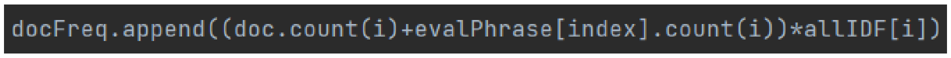


```python
for i in cutoff:
    allIDF[i] = calcIDFWord(i, lowerAllTok)
```

(b) Passing vocabulary

Figure 1: IDF calculation

inverse document frequency, we want terms that appear in more documents to hold a lesser weight. When vectorising our documents for training, for each word in our vocabulary, we will multiply the number of times that term occurs in the document, by the previously calculated IDF value for it, giving us our TF-IDF value for that term in the document (See figure 2). As well as implementing the TF-IDF



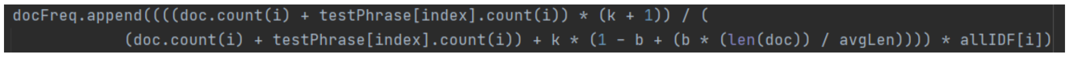
```python
docFreq.append((doc.count(i)+evalPhrase[index].count(i))*allIDF[i])
```

Figure 2: Code snippet for the TF-IDF calculation

algorithm to add weighting for the features, I alternatively implemented BM25, another normalisation algorithm that takes the length of each document into account. This means that shorter documents are not penalised as much for not having as many features as a longer document, for example. In theory, this should work more effectively than the standard TF-IDF algorithm, as it adds the extra functionality of the document length, as well as using the term frequency and document frequency. As with TF-IDF, we initially calculate the IDF value for each word and store them in a dictionary for use in the calculation. When vectorising our training data to pass into the Naive Bayes algorithm, we use the following formula to normalise each token:

$$BM25(t,D) = IDF(t)\frac{f(t,D)\times(k_1+1)}{f(t,D)+k_1(1-b+b(\frac{len(D)}{avLen}))}$$

Where t is the term, D is the document, $k_1 = 3$ and $b = 0.5$ are constants, $len(D)$ is the length of the document, and $avLen$ is the average length of all the documents in the training set. The code that implements this can be seen in figure 3.



```python
docFreq.append((((doc.count(i) + testPhrase[index].count(i)) * (k + 1)) / (
            (doc.count(i) + testPhrase[index].count(i)) + k * (1 - b + (b * (len(doc)) / avgLen)))) * allIDF[i])
```

Figure 3: Code snippet for the BM25 calculation

3 Final Feature Selection

Throughout the implementation process of this classifier, I tested the SKLearn implementation of Naive Bayes with many different feature sets. The main choices that had to be made were: how to tokenise words, whether I extract compositional phrases by using PoS and constituency parsing or by using frequently occurring n-grams (and how many words to use for these n-grams), whether to normalise the word vectors by using TF-IDF or BM25, and what cut-offs to use for my frequency distribution.

I initially tested different cut-offs for the frequency distribution, splitting my tokens by whitespace and using PoS with TF-IDF as my normalisation calculation. I quickly found that this varied highly in accuracy, with results varying between 77.7% and 81.3%, with the optimal frequency cut-off to be higher than 45 and lower than 650 instances. When I implemented a different, more reliable form of tokenisation in stemming, I tested the different cut-offs, shown in table 1

Cut-offs	Accuracy
50, 900	79%
60, 900	78.5%
70, 800	78.5%
80, 850	78%
90, 850	77.5%
75, 850	78.5%

Table 1: Accuracy experiments when using stemming with PoS

When we use PoS with lemmatisation to tokenise, we get the results seen in table 2, where we can see that the results are consistently much less accurate, so we can conclude that this is not an effective method for this task.

Cut-offs	Accuracy
85, 850	73.25%
90, 850	72.75%
80, 850	74%
70, 800	73.75%
75, 850	74%

Table 2: Accuracy values when using lemmatisation and PoS

I then tested a different way of extracting compositional phrases, being frequently occurring n-grams, where n=2, n=3, or n=4. I tested the same cut-off values and used stemming to tokenise the words, so all that has changed from table 1 is the method for extracting compositional phrases. Results can be seen in table 3. When using n=5, I get the same results as when I use n=4. From table 3, we

Cut-offs	Accuracy(n=2)	Accuracy(n=3)	Accuracy(n=4)
50, 900	79%	79.5%	79.5%
60, 900	79.75%	78.75%	79.5%
70, 800	80%	82.75%	82.75%
80, 850	79.5%	81%	81.75%
90, 850	78.75%	80.25%	81.25%
75, 850	80%	81.25%	80.5%

Table 3: Accuracy experiments when using stemming with n-grams

can see that using n=3 and n=4 gives a consistently better performance than using n=2, where the only set of cut-off values that give a better performance is 60 and 900. Therefore, when using n-grams, I will be using n=3 and n=4 for all subsequent tests.

Final tests using the TF-IDF algorithm for normalisation was to use lemmatisation and n-grams where n=3 and n=4 on different cut-off values. Results can be seen in table 4.

Cut-offs	Accuracy(n=3)	Accuracy(n=4)
85, 850	81.75%	80.5%
90, 850	81.25%	80.75%
80, 850	82.25%	81.75%
70, 800	82.75%	81.75%
75, 850	83%	82.25%

Table 4: Accuracy values when using lemmatisation and n-grams where n=3 and n=4

From the above results, we can see that the method with the best performance, was by using lemmatisation and n-grams, where n=3, with a cut-off of 75 and 850 instances, with an accuracy on the evaluation set of 83%. If we then use our other normalisation algorithm, BM25, with $k = 3$ and $b = 0.5$, while still using lemmatisation to tokenise our words, and n=3 n-grams for extracting phrases, we get the results seen in table 5.

Cut-offs	Accuracy
85, 850	82.75%
90, 850	81.25%
80, 850	82.5%
70, 800	82.5%
75, 850	83.5%

Table 5: Accuracy values when using lemmatisation and n-grams where n=3, with the BM25 algorithm

All of the accuracy values in this section are using the SKLearn MultinomialNB() model and using my evaluation set. I will test my implementation of Naive Bayes in the next section, using my final feature set and the test set. When testing the final feature set gained by using BM25, n-grams where n=3 and a cut-off of 75 and 850, on the test set, using SKLearn, I got an accuracy of 82.75%

4 Naive Bayes

Naive Bayes is the algorithm that we use to perform the sentiment analysis classification on our dataset, to determine whether a given document contains a negative review, or a positive review. The simplicity of this algorithm means it generally runs much faster than other methods, such as neural networks, as we take each token independently from all others, meaning our probabilities of each term being in the document given the class, are all multiplied together as if they were independent, where in reality, we know this is not necessarily the case.
The Bayes Formula is as follows:

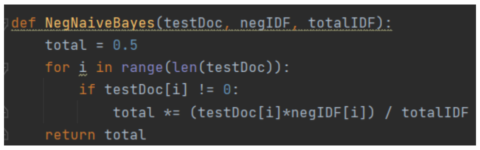
$$P(class|content) = \frac{P(content|class) \times P(class)}{P(content)}$$

After training our Naive Bayes algorithm on the training data, we pass in each testing document, and we separately calculate the probability of the positive class and the negative class given the document, and whichever has the higher probability, we assign that class to the document.
Since we are comparing the probability values for the positive and negative values, we can omit the $P(content)$ from our calculation since it is identical for both calculations, and in our case, since the data split between positive and negative is 50/50, we can also omit the $P(class)$ element from our calculation, leaving just the calculation of $P(content|class)$. With just this calculation remaining, we are left with a calculation of:

$$P(content|neg) \geq P(content|pos) \Rightarrow Negative$$

and vice versa. For this implementation, I am still using the normalised values that we calculated for each document, giving different weights to features of differing importance in order to gain more insight

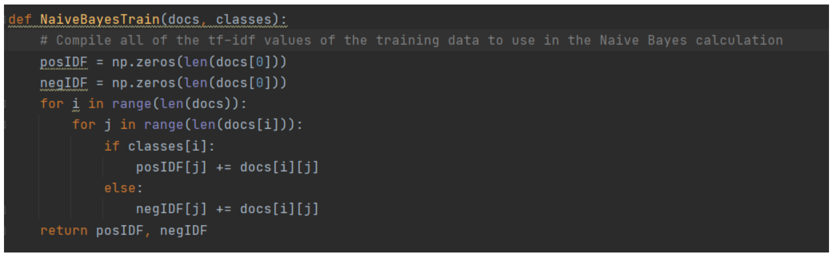
and a higher accuracy than when we just use word counts. The code for the probability calculation of the negative class can be seen in figure 4.
The intuition for the calculation of $P(content|class)$ is to take the BM25 value of the given term in the corresponding document, and divide that by the sum of the BM25 values over the training set of that class. This still ensures that terms with a higher frequency in that document, and higher BM25 values, are given more weight for the calculation.



```python
def NegNaiveBayes(testDoc, negIDF, totalIDF):
    total = 0.5
    for i in range(len(testDoc)):
        if testDoc[i] != 0:
            total *= (testDoc[i]*negIDF[i]) / totalIDF
    return total
```

Figure 4: The calculation for the probability of the negative class

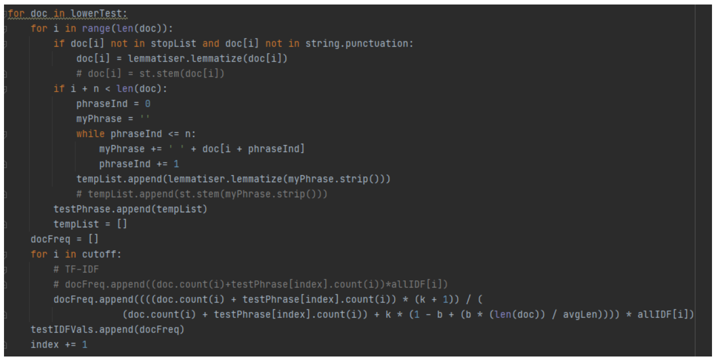
The BM25 values for each token in the document is calculated before being passed into the Naive Bayes function, using the IDF values calculated in the code from figure 1. In order to get the probability of the word given the class, I summed all of the BM25 values of the positive and negative documents separately, and divided the values from the test documents by the corresponding total, which is shown in figure 4 as totalIDF. The calculation for the BM25 values of the overall set of positive and negative documents can be seen in figure 5.



```python
def NaiveBayesTrain(docs, classes):
    # Compile all of the tf-idf values of the training data to use in the Naive Bayes calculation
    posIDF = np.zeros(len(docs[0]))
    negIDF = np.zeros(len(docs[0]))
    for i in range(len(docs)):
        for j in range(len(docs[i])):
            if classes[i]:
                posIDF[j] += docs[i][j]
            else:
                negIDF[j] += docs[i][j]
    return posIDF, negIDF
```

Figure 5: The training algorithm for Naive Bayes

In figure 5, I initialise the positive and negative BM25 values as a list of zeros that is the length of the vocabulary. I then iterate through all documents in the training data, and add the BM25 value for that word to its corresponding class, giving me a final weight for each word in each class to test with. I tested this implementation of Naive Bayes on the exact same feature set that I used at the end of section 3. I first vectorised all of my training and testing documents, using code in figure 6.



```python
for doc in lowerTest:
    for i in range(len(doc)):
        if doc[i] not in stopList and doc[i] not in string.punctuation:
            doc[i] = lemmatiser.lemmatize(doc[i])
            # doc[i] = st.stem(doc[i])
        if i + n < len(doc):
            phraseInd = 0
            myPhrase = ''
            while phraseInd <= n:
                myPhrase += ' ' + doc[i + phraseInd]
                phraseInd += 1
            tempList.append(lemmatiser.lemmatize(myPhrase.strip()))
            # tempList.append(st.stem(myPhrase.strip()))
        testPhrase.append(tempList)
        tempList = []
    docFreq = []
    for i in cutoff:
        # TF-IDF
        # docFreq.append((doc.count(i)+testPhrase[index].count(i))*allIDF[i])
        docFreq.append((((doc.count(i) + testPhrase[index].count(i)) * (k + 1)) / (
                    (doc.count(i) + testPhrase[index].count(i)) + k * (1 - b + (b * (len(doc)) / avgLen)))) * allIDF[i])
    testIDFVals.append(docFreq)
    index += 1
```

Figure 6: Code to vectorise my test set

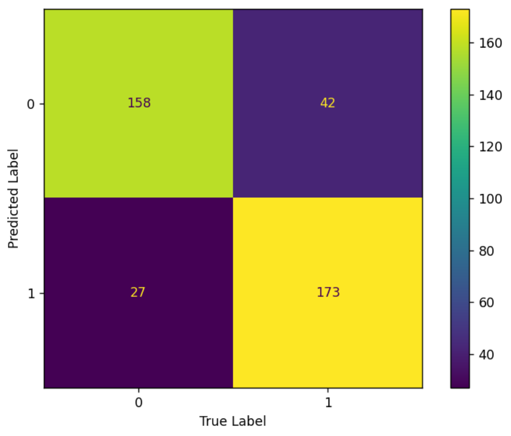
In this code, I use my existing vocabulary that I extracted from my training data, including my compositional phrases. I Extract all my compositional phrases from the test documents, store them in the testPhrase list, and then I iterate through my vocabulary (the cutoff dictionary, which includes all tokens and phrases within the cut-off parameters mentioned in section 3), and add the BM25 scores for each word in cutoff to a list of values to represent that document. Using this vectorisation, and the training values for the positive and negative classes calculated in figure 5, I calculate the probability of each class given my vectorised document (See figure 4) and predict a class. I tested the most effective methods that we saw from the tables above. Results can be seen in table 6, where BM25 is being used as the normalisation algorithm, and for any n-grams, n=3.

	n-grams, lemmatise, cut-off=75, 850	n-grams, stem, cut-off=70, 800	PoS, stem, cut-off = 50, 900
Accuracy	82.75%	81.75%	80%

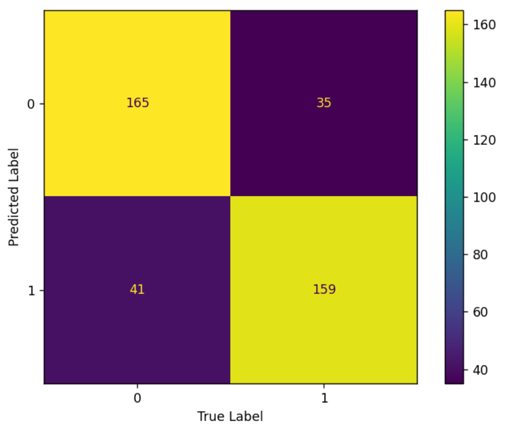
Table 6: Accuracy values from my implementation of the most effective methods

From table 6, we can see that a feature set that uses lemmatisation to tokenise, n-grams where n=3 to extract phrases, with cut-off values of 75 and 850 were the most effective, which is the same result as when using SKLearn's implementation, so I will use this for my final feature set to test on the test set.
Using the final feature set, and testing on the test set, I achieve an accuracy of 81%, meaning my implementation is marginally less effective than the SKLearn implementation, but we can see that it still gains an effective understanding of the sentiment of the reviews.

The confusion matrices for both implementations of Naive Bayes, using the test set and the final feature set, can be seen in figure 7, with figure 7a being the matrix for the SKLearn implementation, and figure 7b being the matrix for my implementation.



(a) Confusion matrix for SKLearn



(b) Confusion Matrix for my implementation

Figure 7: Confusion matrices for the best feature set, running on both implementations of Naive Bayes