# NLP Assignment

## Introduction

## Evaluation Data Splits

The data provided is 4000 text files each containing one review. The 4000 files are already split between 2000 positive and 2000 negative. To carry out the data split I first had to decide the ratios for the training, evaluation and test sets. I decided to split the data in an 80:10:10 manner where the training set would make up 80% of the data and the evaluation set would make up 10% and the test set would make up the last 10%. Industry average is about 60/80:10/20:10/20, so I decided to go with 80% for the training set as it would allow mw to train the model with the most information possible. To split the data, I first looped through all the positive reviews and extracted each review and appended it to an array. At the same time, I scraped the score the review gave from the text file name as each file was named in the same format: someNumber\_score.txt. Depending on if the score was less or more than 5 I then appended a 1(for positive) and a 0(for negative) to another “labels” array. I repeated this process for the negative reviews. From here, I now had four arrays; one positive review array, one negative review array, a positive label array and negative label array. From here I concatenated the review arrays and the label arrays. This allowed me to use sklearn’s train\_test\_split function which allows you to split the data in a ratio of your choosing randomly. I first split the data 80:20 and assigned the 80% to and X\_train and Y\_train array and then split the remaining 20% 50:50 to create the eval and test sets. Finally, I had three sets of data: X\_train which contains 3200 reviews, Y\_train which contains 3200 labels corresponding to each of those reviews, X\_eval of size 400, Y\_eval of size 400, and X\_test and Y\_test each of size 400.

## Feature Generation, Dimensionality Reduction and Features Selection

###### Feature Generation

First thing to be done for feature generation was Tokenization, I chose three methods for this: Firstly, splitting words by whitespace, removing punctuation and lowercasing all words. To do this I created a function which looped through each review and first lowercased the entire sentence using the .lower() function. From there it then used regex to remove any punctuation and then finally once that was done the words were split by whitespace. This meant that each “row” of the review arrays now had a list of individual tokens (e.g. [‘the’, ‘red’, ‘car’]). Now that I had these lists of tokens, I could move on to removing stop words, lemmatization and stemming. I created two more functions; firstly remove\_stopwords\_and\_stem and secondly remove\_stopwords\_and\_lemmatize. Both of these functions work similarly, they both use the nltk corpus stopwords and create a set of unique English stop words from this. They then both remove the word ‘not’ from this set as negation will be very important in sentiment analysis. Then from there the stemmer uses nltk’s Lancaster Stemmer and loops through each token in each review and stems it if it is not in the stop words set. If the token is in the set of stop words it is ignored and removed from the array. The lemmatizer uses nltk’s WordNet Lemmatizer to do the same and lemmatize each token and remove it if it is not in the stop word list.

Finally, this means that I was now able to get raw tokens, stemmed tokens without stopwords, and lemmatized tokens without stopwords or any of the three combined.

I then had to create a cut() function as the number of features was already become unwieldy. This function works by firstly a list of all the words in all the reviews, duplicates included. This then allows me to use nltk’s FreqDist function to create a frequency distribution over all the words. I then get the length of the frequency distribution which returns the count of all unique words, which I then use to get an integer which represents the percentage of words I want to cut. I do this by multiplying the count of unique words by a certain decimal (e.g. 0.05 for 5%). From there I use the most\_common(x) method which returns the ‘x’ most common tokens. However, this is returned as a tuple (token, freq) so I included some list comprehension to parse out just the token to be stored in a list. I then repeat this for the least common tokens and add them to the list. From there, using that list, I loop through each review and remove any tokens which I find match a token in the list. I now have a function which can remove x% of the tokens from the top and bottom of the frequency distribution.

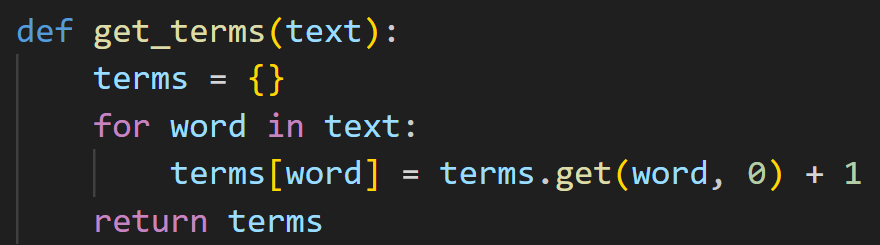
Next was extracting compositional phrases. Firstly, I started with N-grams, I created a function which is passed an array of tokenized reviews and n which represents the number of tokens in each n-gram. The function loops through each tokenized review and uses nltk.util’s ngrams function to produce a list of n-grams dependant on the ‘n’ passed in to the function. This means if it is passed [‘The’, ‘red’, ‘dog’, ‘runs’], 2 it will return [(‘The’, ‘red’), (‘red, ‘dog’), (‘dog’, ‘runs’)].

I now had a function to generate n-grams of any size so I moved on to parts of speech tagging and constituency parsing. The goal with this was to be able to tag each token which would then allow me to parse each and create a parse tree representing noun phrases, verb phrases, etc… This would then allow me to extract the noun phrases and use those as features further down the line. To do this I created a function which firstly uses nltk’s pos\_tag() method which allowed me to tag each of my tokens so that a token was now represented as a tuple (‘DT’, ‘The’) where DT represents determinant and ‘The’ is the token. From there I then declare a grammar representing a noun phrase which I use in line with nltk’s regexpparser function which creates a parse tree for each review. I then feed these parse trees into one last function which extracts the noun phrases. It uses list comprehension to remove the tags and separate tokens by a space. It then appends each noun phrase present in a specific review to an array row representing that review.

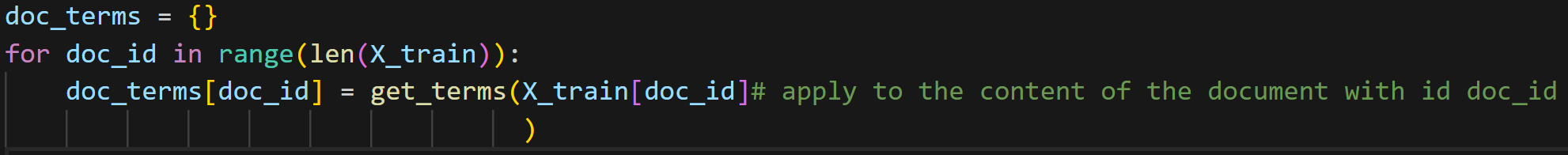
Finally, I now have two functions to extract compositional phrases, one which extracts n-grams and another which extracts noun phrases. I can now add these on to my list of features and calculate TF-IDF for each of these feature sets.

###### Normalise

Now that I have tokens, n-grams, and noun phrases extracted from the reviews I can use these to get TF-IDF scores for each. This will allow me to have a matrix of scores which I can feed into a naïve bayes models to predict on the evaluation set with. I calculate TF-IDF scores for each of these features separately, normalize them and then combine them at the end. I have five functions which I use to calculate the TF-IDF scores, I will explain them sequentially:



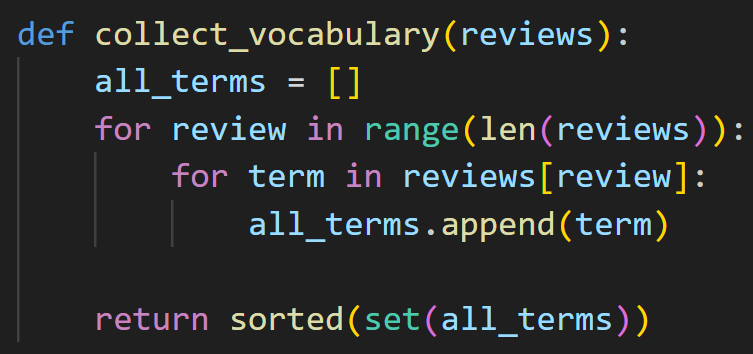
This function is used to create a dictionary to store each of the terms/tokens in a review and its corresponding count of appearances in that review.



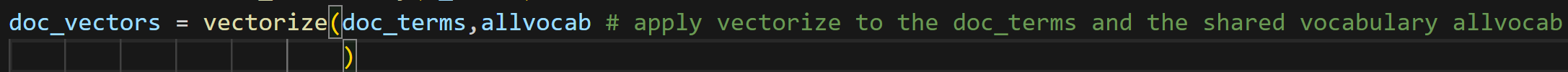
I call this function in a loop which loops through all of the reviews in the given set (here being X\_train). This means that once this loop has finished running, I have a dictionary which holds the counts of each token for each specific review as a dictionary.

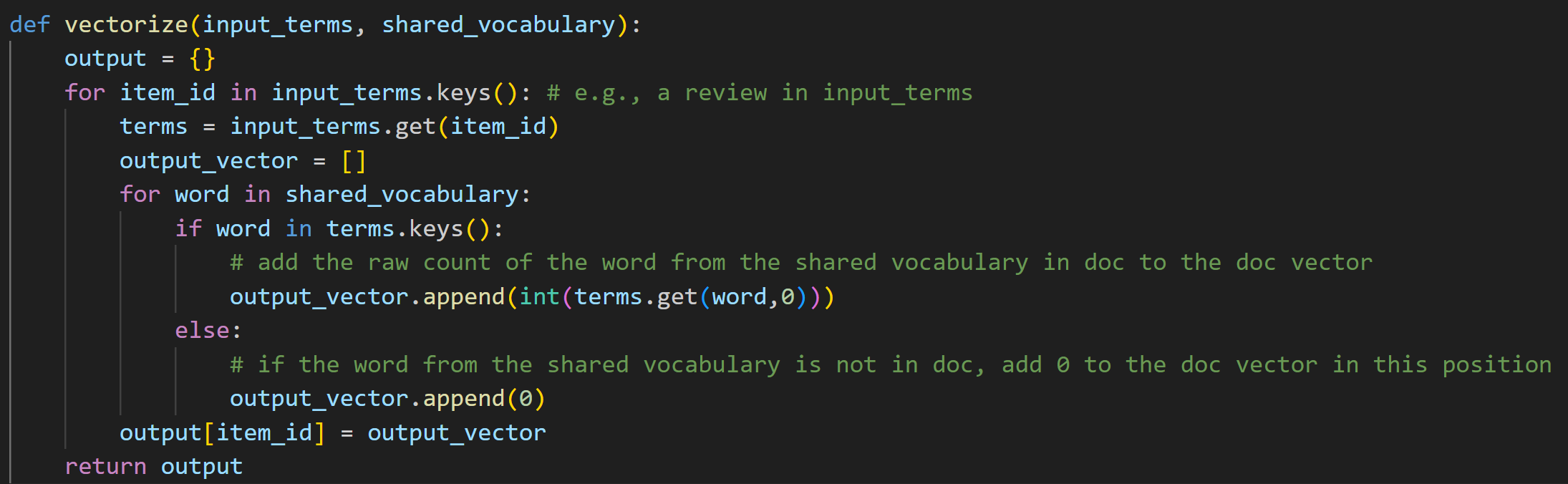
Once that dictionary is created, I next collect all the tokens from the set:





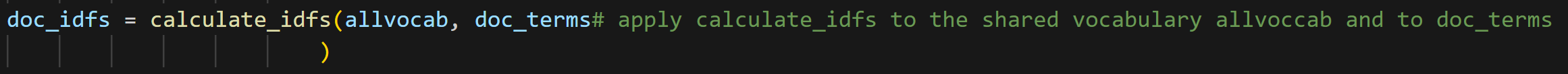
This function works by retrieving every term/token in the given set of reviews and returning a sorted set of these tokens. This gives us a list of each token uniquely which will be used in following functions to calculate term frequencies and inverse document frequencies.

Next, I then need to vectorize each of these review dictionaries so they can be fed into the model. 



This code takes the dictionaries and the shared vocabulary. It then loops through each review and retrieves the dictionary for the relevant specific review. It then loops through each token/word in the shared vocabulary and checks if it is present in that review by looking at the keys(which are tokens) and comparing. If that token is present, then it appends the token count to the end of the output vector. If the token is not present, then it appends a 0 to the vector. This ensures that the review vectors are all the same length (this being the length of the vocabulary) so it can be used in the model.

We now have vectors with counts for each review, we now need to calculate the inverse document frequencies so we can get TF-IDF values. These frequencies represent how important a token is with respect to all the reviews and allows us to add or remove importance from terms which appear a lot across reviews or terms which don’t appear as much across the reviews.



A screen shot of a computer code

Description automatically generated

###### Features Selection

All Features:

* Tokenization (split by whitespace and removes punctuation and turns everything into lower case.
* Remove stopwords and stem
* Remove stopwords and lemmatize
* Cut (dimensionality reduction)
* N-grams
* POS and constituency parsing to extract nounphrases
* TF-IDF

|  |  |  |
| --- | --- | --- |
| Features Used: | Evaluation Score: | Notes and Comments |
| Tokenization with TF-IDF | 0.8475 | 4 min 6.1s |
| Tokenization, remove stopwords and stem with TF-IDF | 0.815 | 3 min 10.8s |
| Tokenization, remove stopwords and lemmatize with TF-IDF | 0.8325 | 4 min 39.0s |
| Tokenization, remove stopwords, stem and lemmatize with TF-IDF | 0.8125 | 3 min 53.3s |
| Tokenization, remove stopwords and lemmatize, cut top 0.5% and bottom 0.5% tokens, TF-IDF | 0.8225 | 3 min 34.7s |
| Tokenization, remove stopwords and lemmatize, N-Grams, TF-IDF, Normalized(minmax) | No result | 20+mins  Abandoned as computation too long |
| Tokenization, remove stopwords and lemmatize, N-Grams, cut top 5% and bottom 5% of tokens, TF-IDF, Normalized(minmax) | 0.7475 | 17mins 44.1s |
| Tokenization, remove stopwords and lemmatize, Extracting nounphrases, TF-IDF | 0.83 | 7mins 19.9s |

## Naïve Bayes

## BERT