# Applied Data Science
# Lab 3 Report

Eklavya Sarkar

November 30, 2018

## Introduction

This assignment builds on the two previous ones, requiring again our skills in efficient data ingestion, wrangling, cleaning, visualisation, etc. along with Machine Learning algorithms implementation. This time the goal is to build two text classifiers with different approaches and algorithms, in order to closely understand the factors which influence the accuracy of the sentiment analysis.

## Code Methodology

### Libraries

Similar to the previous coursework, I used `Pandas` for data wrangling, `NumPy` for a few specific functions such as `np.arange`, and `Matplotlib` for data visualisation. The `Re` library was employed to construct regular expressions for tokenisation, and `stop_words` library for an independent bag of stop words. Finally, `sklearn` was used for a number of functions, such as model selection, feature extraction, metrics, and Machine Learning algorithms like Naive Bayes and Support Vector Machines. Notably `TfidfTransformer`, `CountVectorizer`, `MultinomialNB`, `LinearSVC` were the key methods in solving this assignment.

I specifically chose not to use the `NLTK` library, or `sklearn`'s integrated tokenisation method, but to refine my implementation of a tokenisation method built for a previous assignment. This allowed me to have a closer understanding of my code's overall process.

### Data Ingestion and Wrangling

I used `Pandas` to read in the source data from the `.csv` files with a `encoding='ISO-8859-1'` parameter for correct data ingestion, and a `usecols=[0, 1]` one, as only the first two columns contained the relevant information.

### Data Cleaning

The first step was to remove any unnecessary white space, from both the left and right sides, which can be done on a whole `Pandas DataFrame` column by using the `str.strip()` method. Similarly, all words in a tweet were converted to lower-case with `str.lower()`.

The second step was to tokenise each tweet using a regular expression, which caught and removed all characters except numbers, alphabets, semi-colons, hashes, underscores, dashes, at-signs, and front and back-slashes. I decided to not limit my selection to purely alphanumeric characters, as I felt ASCII emoticons, such as `:)` or `T_T` also conveyed a distinct sentiment, and thus could perhaps be a significant factor in the sentiment analysis. Any other characters, punctuation or letters not mentioned above were removed.

```
cleanText = re.compile(r'[^0-9a-zA-Z
    :\-|\/\\_@#]')
```

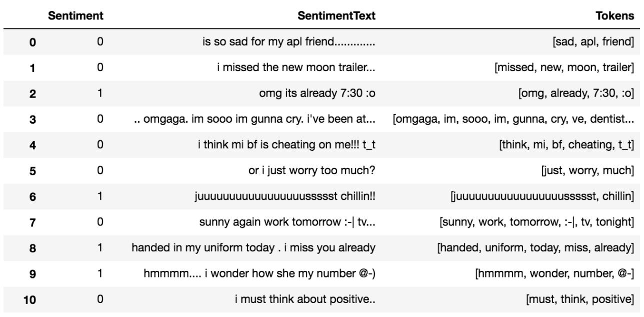
Listing 1: Regular expression tokenisation

The final step was to remove tokens which were part of the common stop words or composed of a single character. One could go further and remove words with three characters or less, which would decrease the overall computation time, however I wanted to make sure to include two-character emoticons, and thus restricted my word-length admission to only higher than 1.

```
wordList = []
for t in tokens:
    if t not in stop_words and len(t)>1:
        wordList.append(t)
```

Listing 2: Stop-words removal

By having the above in methods, I was able to tokenise and filter each tweet into a list of separated words, and have it as another column in my `Pandas DataFrame`.



| | Sentiment | SentimentText | Tokens |
|---|---|---|---|
| **0** | 0 | is so sad for my apl friend............ | [sad, apl, friend] |
| **1** | 0 | i missed the new moon trailer... | [missed, new, moon, trailer] |
| **2** | 1 | omg its already 7:30 :o | [omg, already, 7:30, :o] |
| **3** | 0 | .. omgaga. im sooo im gunna cry. i've been at... | [omgaga, im, sooo, im, gunna, cry, ve, dentist... |
| **4** | 0 | i think mi bf is cheating on me!!! t_t | [think, mi, bf, cheating, t_t] |
| **5** | 0 | or i just worry too much? | [just, worry, much] |
| **6** | 1 | juuuuuuuuuuuuuuuussssst chillin!! | [juuuuuuuuuuuuuuuussssst, chillin] |
| **7** | 0 | sunny again work tomorrow :-\| tv... | [sunny, work, tomorrow, :-\|, tv, tonight] |
| **8** | 1 | handed in my uniform today . i miss you already | [handed, uniform, today, miss, already] |
| **9** | 1 | hmmmm.... i wonder how she my number @-) | [hmmmm, wonder, number, @-] |
| **10** | 0 | i must think about positive.. | [must, think, positive] |

Figure 1: Tweets before and after tokenisation

### Data Exploration

I first explored the data to observe the distribution of the positive and negative tweets. As shown below, the data is relatively balanced and un-skewed towards either type of tweet.

| | **Positive** | **Negative** |
|---|---|---|
| **Counts** | 554470 | 494105 |
| **Normalised** | 0.53 | 0.47 |

Table 1: Tweets sentiments distribution

I also looked at the number of words in each tweet and calculated the mean length of all tweets, which is 7.959, and the standard deviation of the lengths which is 4.00.

### Data Analysis and Modelling

The modelling for Task 1 can be done either manually, by creating a frequency table, likelihood table, and word-bag dictionary, or else simply through an `sklearn` method. In my code, I chose to do it both ways, and obtained very similar results. Task 2 can similarly be either manually or through `sklearn`, however this time I chose to only do it the latter way, which is why the in-depth description of my code below is also for `sklearn`.

## Code Development

### Task 1

Implementing Naive Bayes with `sklearn` is pretty simple if one understands the process, which can be divided into three big steps. Firstly, one has to transform the data into occurrences using the `CountVectorizer()` method, which become the features of our model, i.e. the columns. This essentially converts a collection of text documents, in our case the `Pandas DataFrame` of lists of tokenised words, to a matrix of token counts. The method produces a sparse matrix of type `‘<class ‘numpy.float64’>’`, in which most of the elements are `0`s and a few `1`s. In our case the tokenised words `DataFrame` would be the one converted to a sparse matrix.

```
corpus = [
  'This is the first document.',
  'This document is the second document.',
  'And this is the third one.',
  'Is this the first document?'
]
```

Listing 3: Sample list of strings

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 2 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

Table 2: Sparse matrix of strings post vectorisation

Secondly, we use the `TfidfTransformer()` method to create an inverted-index, which ranks the words according to their importance which increases proportionally to the number of times a word appears in the document but is also offset by the frequency of the word in the corpus.

Finally, we can split the data into the training and testing sets according to the specified `4:1` ratio using the `train_test_split()` method, with a `train_size=0.8`, `test_size=0.2`, and `random_state=6` parameters, so that the split is of the same ratio, but not of the same rows every time.

Once we have extracted the features, ranked according to importance, and split into training and testing sets, we can simply use `sklearn`'s `MultinomialNB()` algorithm to fit the data, evaluate it on the testing data, and print the confusion matrix.

| | | **Predicted Label** | |
|---|---|---|---|
| | | **Negative** | **Positive** |
| **True Label** | **Negative** | 77375 | 21744 |
| | **Positive** | 24093 | 86503 |

Table 3: Confusion matrix without normalisation

We can normalise the data:

| | | **Predicted Label** | |
|---|---|---|---|
| | | **Negative** | **Positive** |
| **True Label** | **Negative** | 0.78062733 | 0.21937267 |
| | **Positive** | 0.21784694 | 0.78215306 |

Table 4: Confusion matrix with normalisation

We can visualise the confusion matrix with a colour-scale using `Matplotlib`.

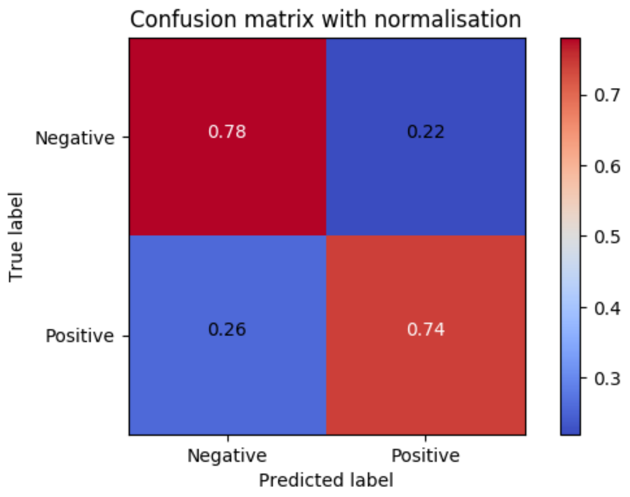


Figure 2: Multinomial naive bayes normalised confusion matrix without prior

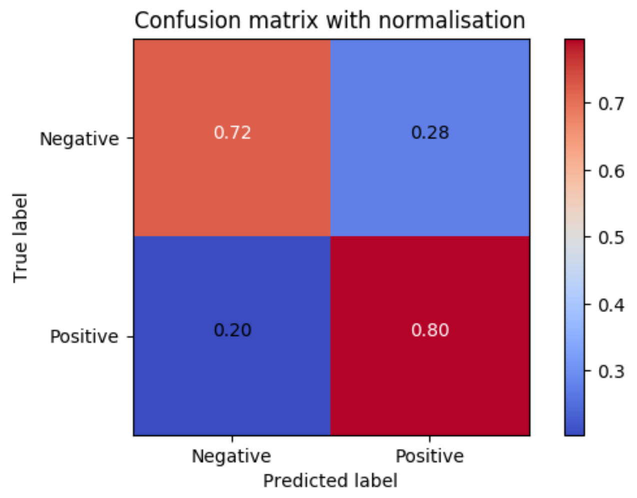


Figure 3: Multinomial naive bayes normalised confusion matrix with prior

| **Score \Model** | **NB w/o prior** | **NB with prior** |
|---|---|---|
| **Training** | 0.936 | 0.934 |
| **Testing** | 0.781 | 0.781 |
| **Mean-Square** | 0.219 | 0.219 |

Table 5: Training, Testing and Mean-Square Error scores of MultinomialNB algorithm with and without a prior

**Task 2**

The key to task 2 was understanding that the Naive-Bayes algorithm treats each word independently, meaning it looks at them individually when calculating the occurrences and frequencies. While this has the advantage of being simple to understand and efficient computationally, it is limited when it comes to subtleties. For example, it does not take into account the position of any of the words, nor does it give importance to words which influence the following ones, such as *not* and *can't*. Overall, the Naive Bayes algorithm seems to perform well to when keywords are essentially the features, but not as well when the words are dependent and their relationship important, which is relevant to sentiment analysis.

Thus, by trying take into account the dependencies of words, we can try to improve the score. The `CountVectoriser` method contains a `n-grams` parameter, which allows us to analyse words as sets, depending on the chosen minimum and maximum words, instead of individual single words. Additionally, `Sklearn` provides Support-Vector Machine algorithms, in our case `LinearSVC`, which yields a better classification for bigger datasets. Thus, by recalculating the counts, then transforming them into an inverted matrix, and then splitting the data, we obtained the following results.

| **LinearSVC: n_grams (min,max)** | **Train** | **Test** |
|---|---|---|
| **(1,2)** | 0.99249 | 0.79292 |
| **(1,3)** | 0.99775 | 0.79574 |
| **(1,4)** | 0.99812 | 0.79589 |
| **(1,5)** | 0.99814 | 0.79508 |
| **(1,6)** | 0.99813 | 0.79439 |

Table 6: The train and test scores with the LinearSVC() method and a count with various ranges of ngrams.

As we can see the score of the test data doesn't necessarily increase with the maximum size of the `ngram_range` parameter, nor do the scores improve significantly with the different ranges. Given the longer computational time taken for each range, it could be argued the bi-gram range is the optimal method when taking into account both the final score and the computational efficiency. Overfitting is also an issue, when we know from the data exploration that the mean length of a tweet is 7.959.

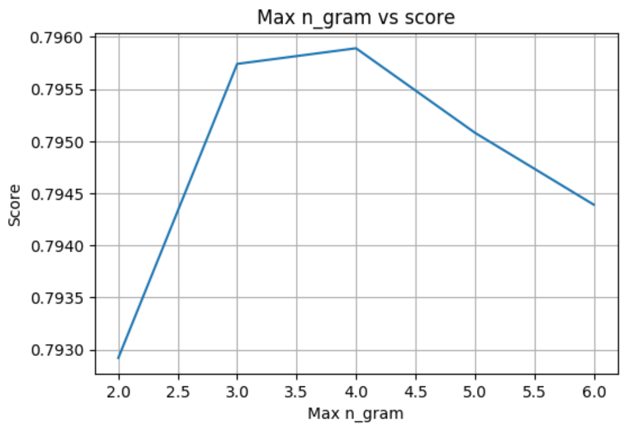


Figure 4: Evolution of training score in function of max ngram value. Note that the scores measures are actually for specific discrete ngram values, but this graph is relating them into a continous line to show their evolution.

## Code Development Issues

With the help of `Sklearn` there were very few technical obstacles in implementing the algorithms, fitting the models, and getting the predicted values.

The core challenge in this assignment was actually conceptual - understanding the process and structure of the Naive Bayes algorithm, and how to it uses a bag of words to make a frequency table, and use it to calculate the sentiment probabilities.

## Noteworthy results

While the difference between the test scores from the Naive Bayes implementation and the Linear SVC is only marginal (relatively speaking), is it definitely better on all counts of n-grams. This fits with our hypothesis that the the test score would improve if the algorithm considered sets of words, rather each one independently. We can also see an improvement in the confusion matrix between the multinomial Naive Bayes and LinearSVC's optimal (1,4) ngram range.

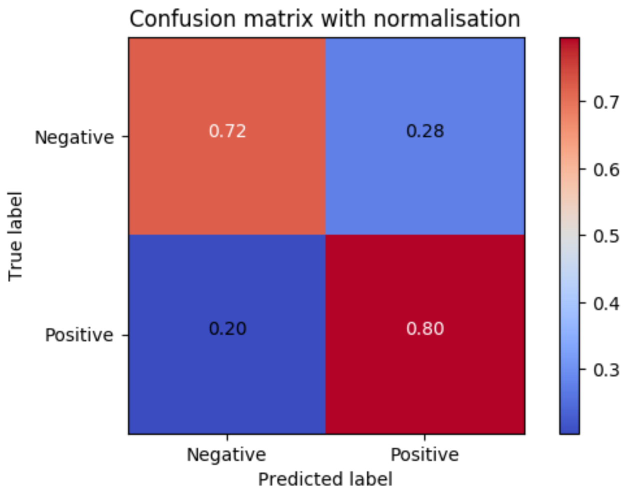


Figure 5: Naive Bayes confusion matrix

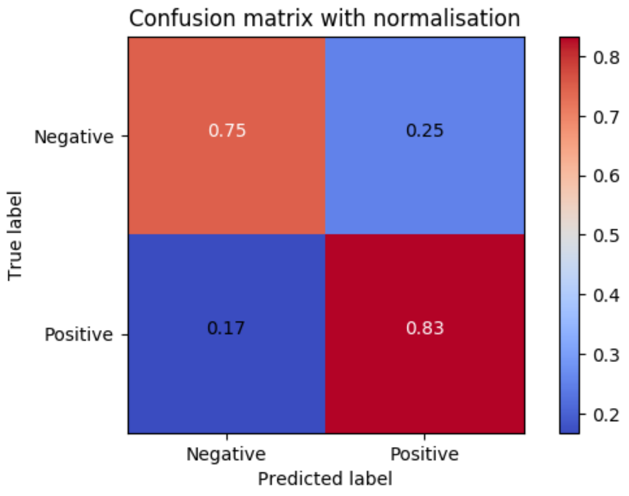


Figure 6: Linear SVC with (1,4) confusion matrix. The matrices of with other ngram values give the same confusion matrix with a maximum difference of ± 0.01.

## Conclusion

This assignment was important in helping us understand and analyse the factors which influence the accuracy of a classifier, and how to interpret the results of a confusion matrix along with test scores. We also learnt about subtleties of the Naive Bayes algorithm, such as the fact that that while it is computationally efficient, it can also quickly become limited for large amount of databases as it only considers words independently. Instead if a count vectoriser which takes into account the dependency is used with a different algorithm, such as Support Vector Machine, one can yield better results.