Machine Learning
Assignment 1: Bayesian Classification

R00159222 - Zachary Dair
zachary.dair@mycit.ie

Extra Information:

The assignment was done in python 3.8, uses the following libraries:

math, pandas, time, sklearn.model_selection, random

Each section of the functionality is split into functions, with a function to run tasks 2-5, and after all function definitions *(Line 188)* We begin running the functions and the K-folds.

There is a somewhat high line count due to my spacing and comments.

Task 1: Splitting and counting the reviews *(Defined Line 10)*

```
def reviewTestTrainSplit(filename)
```

The above function takes in a string (the filename) as an argument, performs basic error checking using a try and except to ensure the file is present.

In the case the file is absent, an error message is displayed.

Otherwise the data is retrieved from the movie_reviews.xlsx file using the pandas.read_excel function, and our columns ('Review', 'Sentiment', 'Split').

Splitting the Dataset: *(Line 23-26)*

The data variable (pandas dataframe) is then split into four separate variables:

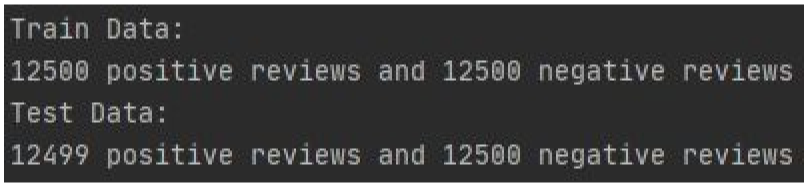
trainData - the contents of the 'Review' column where the 'Split' column is equal to 'train'

trainLabels - the contents of the 'Sentiment' column where the 'Split' column is equal to 'train'

And replicated again for testData and testLabels where the 'Split' value is equal to 'test'

Counting positive/negative reviews: *(Line 30-35)*

Following this we can count the number of labels that are equal to 'positive' and then 'negative' using the len function, and on both trainLabels and testLabels.



```
Train Data:
12500 positive reviews and 12500 negative reviews
Test Data:
12499 positive reviews and 12500 negative reviews
```

We then finally return our four lists.

Task 2: Extract relevant features *(Defined Line 42)*

`def cleanAndSplitReviews(reviewData, minWordLength, minWordCount)`

The above function takes in a pandas series (the review data) as an argument, an integer for the minimum word length and minimum occurrence count for a word.

Pre-Processing: *(Line 44-50)*

Firstly we need to remove non-alphanumeric characters excluding spaces, done using:

`reviewData.str.replace('[^a-zA-Z0-9 \n]', '')` and storing the returned value.

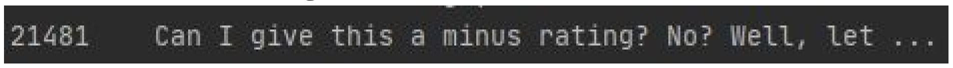
Secondly we convert the strings to lowercase using:

`reviewData.str.lower()` and storing the returned value

I considered converting the string to lowercase, and removing the capitals from the replace to improve timing, however it was slower (0.051 VS 0.048 seconds)

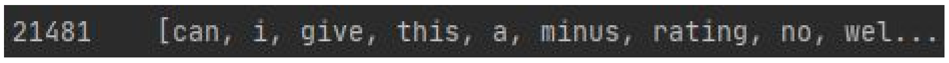
We then split all reviews using space as a delimiter using `reviewData.str.split(" ")`

Before Pre-Processing



```
21481    Can I give this a minus rating? No? Well, let ...
```

After Pre-Processing



```
21481    [can, i, give, this, a, minus, rating, no, wel...
```

Occurrence Counting: *(Line 53-60)*

We use a dict to store the word and their occurrence count in key pairs.

By converting our reviewData to a numpy array we can cycle through each review in a for loop, then cycle through each word of the review's sub-list checking if the word is in our wordCount dict, if it is we check its current value and add one, if not we set it's value in the list to one.

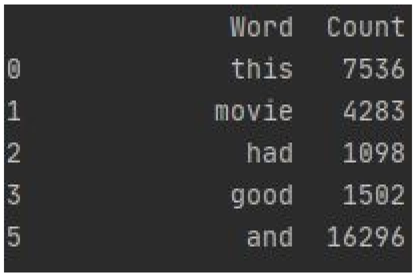
For easy conditional extraction, we create a dataframe for the words, using the dict.items function on our wordCount dict, and using columns: 'Word' and 'Count', using this dataframe we can then extract the words where str.len is greater or equal to the min word length, and the words where the count is greater or equal to the min word count all without iterating through the words. *(Line 63-68)*

Adding a condition to remove all words below a certain length in the for loop prior to counting would reduce the time it takes to complete.

We then return the words column from the dataframe converted to numpy, and the processed reviewData (to prevent us needing to clean and split it again)

Word Occurrence Counts

Here we have the first 5 words, when min word length is 1, and min occurrence is 100



```
              Word  Count
0             this   7536
1            movie   4283
2              had   1098
3             good   1502
5              and  16296
```

Task 3: Count Feature Frequencies *(Defined Line 75)*

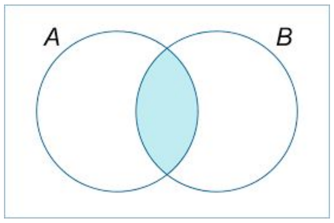
```
def countFeatureFrequencies(reviewSet, wordSet)
```

The above function takes in a pandas series (the newly processed review data) as an argument, and a numpy array containing all the words that matches our task 2 requirements.

This function runs twice, once for the positive reviews and once for the negative ones.

Frequency Counting: *(Line 76-89)*

We convert our review set (all positive or all negative reviews) into a numpy array, and cycle through each review, then using sets and list comparison we extract the intersection of the two lists.



A = Review content as a list of words

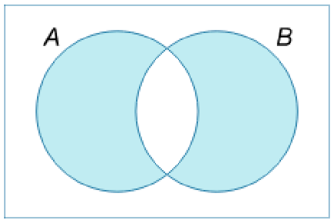
B = Our extracted words from task 2

We then cycle through the common words (intersection) and check if they are present in the keys of the wordCount, if so we increment the count, if not we set the count to 1

Using this intersection method allows us to reduce the iterations of the loop massively as we only need to check the common words instead of every word in the review.

Mapping Un-found Words: *(Line 92-97)*

Using a similar method as above but the exact opposite (the difference) allows us to find all words in the wordset from task 2, that aren't in the word count.



A = The keys of the wordCount dict (all previously counted words)

B = Our extracted words from task 2

We then use the dict.update function to append all unfound words into the wordCount with a count value of zero.

Finally, the dict containing the words and occurrence counts is returned.

Task 4: Calculate feature likelihoods & priors *(Defined Line 104)*

```
def calculateLikelihoodsAndPriors(positiveFF, negativeFF, smoothingVal,
tpCount, tnCount)
```

The above function takes the dicts containing the feature frequencies for both positive and negative reviews (from task3), a smoothing value, two counts for positive and negative reviews.

Likelihood Calculation: *(Line 105-113)*

We loop through each word in the positive feature frequency dict, we use the count value for that word and add the smoothing value, which is then divided by the smoothing value multiplied by the length of the positive feature frequency dict added to the sum of the word counts in the feature frequency dict.

$P[\textit{word in review} \mid \textit{review is positive}]$ & $P[\textit{word in review} \mid \textit{review is negative}]$

Each word likelihood calculated is then stored in a dict in key pairs (word: likelihood)

This calculation is repeated for both positive and negative feature frequency dicts

An evaluation to ensure the likelihoods are correct is to get the sum of the positive, and the sum of the negative likelihoods should both equal 1.

Prior Calculation: *(Line 116)*

The priors, which is the fraction of positive and negative reviews in our training set.

Calculated by dividing the positive review count by the total review count (positive plus the negative review counts), this gives us $P[\textit{review is positive}]$.

$P[\textit{review is negative}]$ is the same but we divide the total review count by the negative review count.

We then return the likelihood dicts for both positive and negative, and the positive and negative priors.

Task 5: Classification *(Defined Line 120)*

```
def classifyReview(testReview, posL, negL, posPrior, negPrior)
```

The above function takes the review to classify as a string, also the dicts containing both positive and negative likelihoods, and the priors for each class.

Pre-Process: *(Line 122)*

As done in task 2, we convert the review string to lowercase, remove any non-alphanumeric characters, and finally convert the string to a list of words.

Classification: *(Line 125-138)*

Using our list comparison we find the intersection of the words in the review and those in our words list, which is now the keys in either of the likelihood dicts.

We can firstly check if no common words were found, in this case the classification is impossible with the data we have, initially I tracked these failed classifications, and they can be seen increasing as the length of the words increase.

Currently these failed classifications are now done at random, having a 50% chance of being correct.

In the case we have found common words, we can then use the likelihoods, we get the sum of the values in log form for positive and negative likelihoods.

Then using logical comparison, comparing the positive log likelihood minus the negative log likelihood and then the log of negative prior minus the log of the positive prior to see which is bigger, if the result of the likelihood calculation is bigger, the classification is positive, otherwise it’s negative.

The classification is then returned and appended to a list, so that it may be compared with the labels.

Task 6: K-Fold Cross Evaluation: *(Line 200-253)*

```
def trainAndRunModel(minLen, train_Data, test_Data)
```

```
for train_index, test_index in kf.split(trainData)
```

The above function encompasses tasks 2-5, using the train data, and minimum word length and test data it runs using tasks 2-5 and returns a completion time, number of correct, number of wrong, overall accuracy and confusion matrix for each model.

The above for loop is what 'drives' the k-fold cross validation.

K-Folds: *(Line 192-219)*

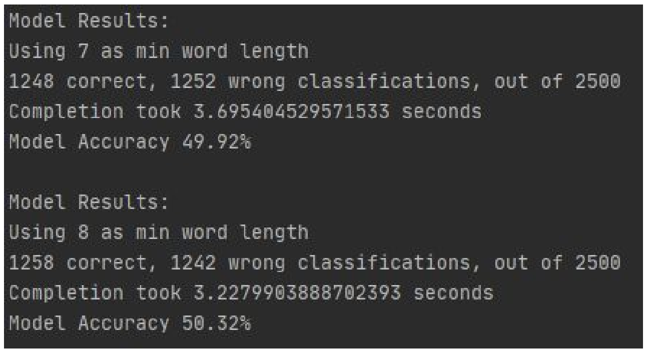
Using KFold with n_splits 10, we can then split our training data into k-folds of size 2500, where each can be used with our previous code in order to train separate models, where the minimum word length increments with each k-fold, starting at 1 and ending at 10.

Classification Accuracy: *(Line 182)*

Using the results of our classifications, the number of correct, and number of wrong classifications we can get the percentage of correctly classified reviews, which is the number of correct divided by the total (number of correct + number of wrong)

This value is returned from the trainAndRunModel in the variable acc.

Below is the accuracy for two models:



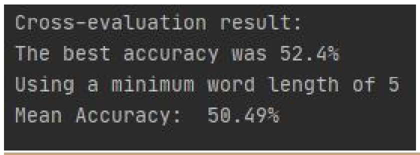
```
Model Results:
Using 7 as min word length
1248 correct, 1252 wrong classifications, out of 2500
Completion took 3.695404529571533 seconds
Model Accuracy 49.92%

Model Results:
Using 8 as min word length
1258 correct, 1242 wrong classifications, out of 2500
Completion took 3.2279903888702393 seconds
Model Accuracy 50.32%
```

Mean Accuracy: *(Line 232)*

For each accuracy value, it is then added to the accuracy total which is divided by the number of models, or in our case the length of the model results list, to give us the mean.

Since adding the K-fold functionality, the mean accuracy tends to stay around 48-52%



```
Cross-evaluation result:
The best accuracy was 52.4%
Using a minimum word length of 5
Mean Accuracy:  50.49%
```

Model Comparison: *(Line 222-227)*

Using the list of our model results, we cycle through these results in a for loop, in order to identify the best performing model in terms of accuracy, we then extract the word length that corresponds to the highest accuracy in our results, by doing this in code, it allows will cause the results to differ, as there is a slight randomness, from the k-fold data splits, and the classification in the case no common word is found.

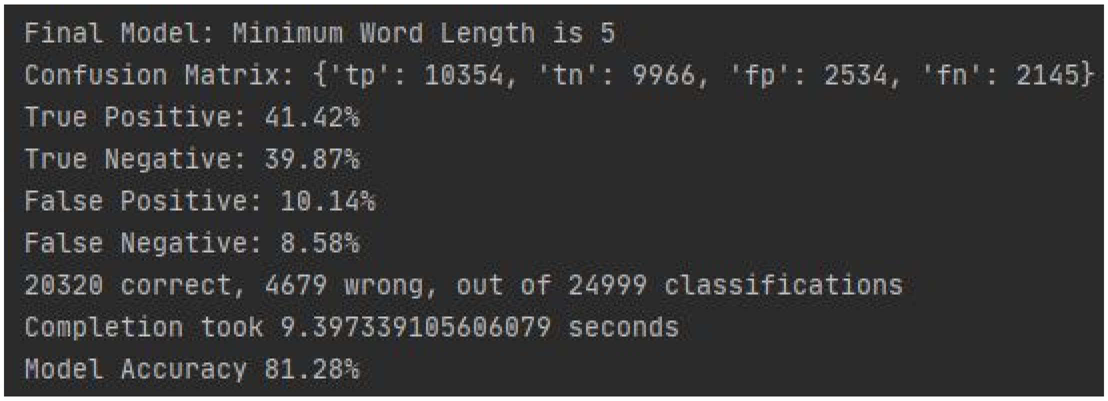
Optimal Model: *(Line 236)*

From the result of our model comparison, which can be seen in the cross-evaluation result image above, we see that in this case the best accuracy was 52.4% using a minimum word length of 5.

We then run one final model, using this 5 min length, this model runs on the full train dataset, and our unseen test data.

Final Evaluation: *(Line 235-253)*

Once our final optimal model has finished training and classifying the data, we are given our results.



```
Final Model: Minimum Word Length is 5
Confusion Matrix: {'tp': 10354, 'tn': 9966, 'fp': 2534, 'fn': 2145}
True Positive: 41.42%
True Negative: 39.87%
False Positive: 10.14%
False Negative: 8.58%
20320 correct, 4679 wrong, out of 24999 classifications
Completion took 9.397339105606079 seconds
Model Accuracy 81.28%
```

The confusion matrix, created in a dict format, and displayed. (tp = true positive, tn = true negative, fp = false positive, fn = false negative)

The percentages for each are also displayed.

We also have the number of correct, number of wrong and total number of classifications.

Timing is also important and we display this result as well, this was done for each model to improve and optimize the timing as much as possible.

And finally we can see the model's overall accuracy, this is the percentage of correctly classified reviews.