2024

Machine Learning Assignment 1

RESULTS AND REPORT

EMMETT FITZHARRIS R00222357

Task 1

I start by reading the data from the excel file in read_data()

```
# Read data from the Excel file.
# The file is large, and this process is slow
# since I will be running this a number of times, I decided to cache the data frame
def read_data():  1 usage   Emmett *
    cache_path = os.path.join(cache_folder, main_cache_filename)
    if os.path.exists(cache_path):
        with open(cache_path, 'rb') as f:
            df = pickle.load(f)
        print('Loaded data from cache\n')
    else:
        df = pd.read_excel(source_filename + file_extension)
        with open(cache_path, 'wb') as f:
            pickle.dump(df, f)
        print('Loaded data from Excel and cached it')
    return df
```

Knowing that I would be running this program many times, I decided to implement a cache system to serialize the data frame, rather than needing to read from the Excel file each time.

Initial read time was approximately 5.05 seconds

```
Execution time: 5.05 seconds
```

After caching, read time is less than 0.13 seconds.

```
Execution time: 0.13 seconds
```

This allows me to iterate more quickly, rather than waiting for the same data to be read again and again. I recorded this result by simply wrapping my main function as follows:

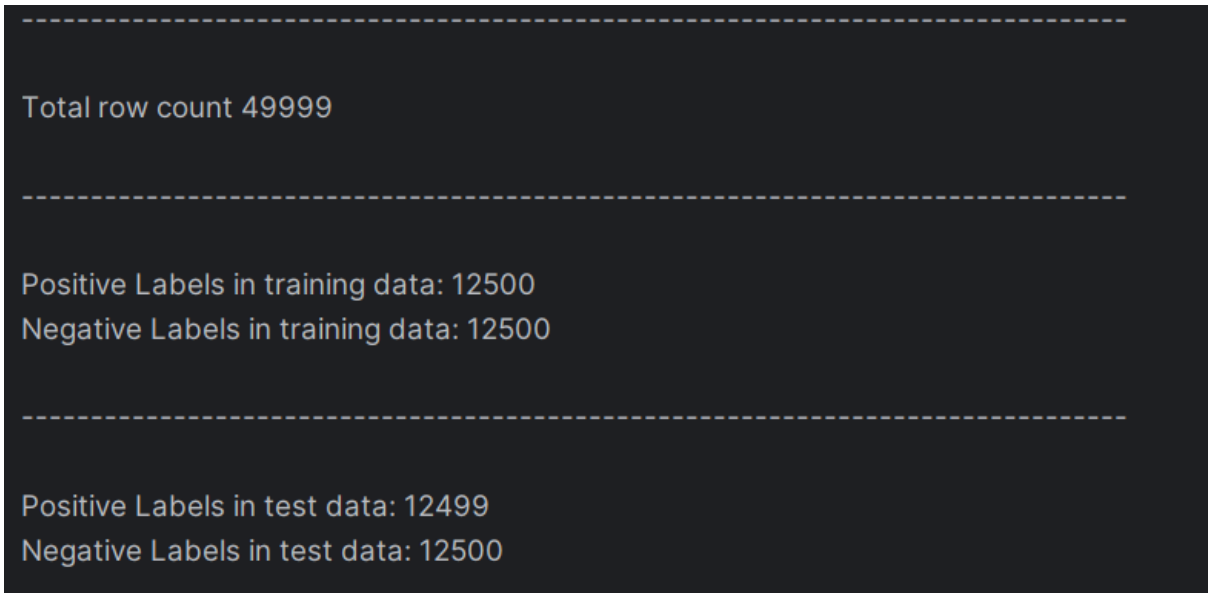
```
def main():  1 usage   new *
    start_time = time.time()
    df = read_data()

    print("Frame size", df.shape[0])
    train, test = split_data(df)


    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Execution time: {elapsed_time:.2f} seconds")
```

This early stage of the program does not have particularly exciting results, but I continue to cache results across the program, which results in significant time saving.

In the task1() function, passing in the data frame. Which splits the data into test and training sets, then prints the counts of positive vs negative labels.



I then bundle the outputs into Pandas data frames and pass back to main.

Task 2

In task two I filter the data frames by minimum_word_length and minimum_word_occurrence.

This is done by filtering out non alphanumeric characters with a pair of reg-ex queries. The data is then made lower case, and each word is split into a list item.

A new data frame is created, by finding those words which meet the above requirements, and running the value count function to count occurrences. This data frame will act as the feature set for model training.

```
[25000 rows x 2 columns]
Number of unique words in training data, longer than 5 characters, which occur more than 50 times: 5758
```

The data appears to be evenly split. The result of these queries varies greatly based on the parameters passed into it. The values I chose at this stage were intentionally high to keep iterations quick:

```
minimumWordLength, minimumWordOccurrence = 5, 50  # these values are intentionally high for testing
```

Task 3

In task 3, we count the number of positive / negative reviews each word occurs in. The function I made for this task requires a loop which takes a long time to run, so I implemented a caching system again to store the result. I was mindful to ensure that it was sensitive to changing minimum word length and minimum occurrence parameters.

At a minimum word length of 10 and a minimum word occurrence of 20, the execution time observed was over 80 seconds:

```
Execution time: 83.73 seconds
```

With the cache in place the program takes less that 7 seconds to run. A reduction in runtime of over a minute. Keep in mind that these parameter values are very high for testing, so the difference will be even more pronounced at normal values, making iterations much more quick.

```
Execution time: 6.75 seconds
```

My results below are intentionally using long words to keep runtime low:

```
################################################################################
                                    Task 3
################################################################################

Loaded data from cache: cache\movie_reviews-train_word_counts_pos-5-50.pkl

Positive train_word_counts_pos
           Word  review_count
0         movie          7460
8         other          5627
41        thing          5338
1         about          5016
5         story          4721
...         ...           ...
5708  casablanca             0
5713     muslims             0
5715       tomei             0
5698      snipes             0
5701      harlow             0

[5758 rows x 2 columns]

--------------------------------------------------------------------------------

Loaded data from cache: cache\movie_reviews-train_word_counts_neg-5-50.pkl

Negative train_word_counts_pos
         Word  review_count
0       movie          8663
41      thing          6974
8       other          5562
1       about          5406
19      watch          4844
...       ...           ...
4394  kolchak             0
2101   brooks             0
2082    larry             0
2084   rachel             0
5707   lionel             0

[5758 rows x 2 columns]
```

The most used words (5 letters and above) are common across both sets, and tend to be descriptive of the industry.

Task 4

In task 4, I calculate some statistics regarding the data. The probability that a word is in a review, given that it is positive, as well as the reverse, also known as the prior probabilities.

```
################################################################################
                                    Task 4
################################################################################

Priors
Training DataProbability. Positive: 0.50, Negative: 0.50

--------------------------------------------------------------------------------

 Number of training Reviews: 25000

--------------------------------------------------------------------------------
```

This turns out to be almost exactly 50/50, which I confirmed by printing the numbers of each type of review.

Following this, I calculate the conditional probabilities, and add them as a new column in my data frame. The conditional probabilities are calculated as follows:

Training Data

Positive

	Word	review_count	probability	conditional_probability
0	performances	1083	0.043357	0.417343
1	entertaining	785	0.031437	0.412575
2	unfortunately	187	0.007519	0.403008
3	particularly	502	0.020118	0.408047
4	cinematography	525	0.021038	0.408415
..	...	...	...	...
66	deliberately	56	0.002280	0.400912
67	unintentional	39	0.001600	0.400640
68	consistently	62	0.002520	0.401008
69	revolutionary	46	0.001880	0.400752
70	specifically	46	0.001880	0.400752

Based on these results we can see that the conditional probabilities, that a word is in a review, given the review has a specific sentiment, are approximately twice the probabilities of a word being in any given review. This makes sense given the 50/50 split of the priors.

Task 5

In task 5 I run a Bayesian classifier algorithm, by calculating the log likelihood of a word being in a review based on sentiment. I use Math.log to calculate the log value from the prior, then for each review, I add the log likelihood of the word being in the review.

This value is calculated for both positive and negative values, and the two are compared. Whichever value is larger is used as the prediction.

Once again, I implement a cache, this time saving the predictions returned by the classifier. The amount of time saved with this cache is very large, as the classifier loops are very expensive to process.

Even with the caching the Bayesian predictor still takes a long time to run. I implemented pooling to make full use of my CPU. This results in processing time being reduced by approximately a factor of 15, as I used all but one of my logical cores.

```
# if the cache is empty, run the prediction task
if train_predictions is None:
    num_processes = multiprocessing.cpu_count()-1  # Use all available CPU cores but 1 to allow the system to perform
    chunk_size = len(training_df) // num_processes
    results = [None] * num_processes

    tasks = [(i * chunk_size, (i + 1) * chunk_size if i != num_processes - 1 else len(training_df), i, training_df,
              train_word_counts_pos, train_word_counts_neg, training_positive_prior, training_negative_prior)
             for i in range(num_processes)]

    with Pool(processes=num_processes) as pool:
        results = pool.starmap(process_task, tasks)

    train_predictions = [pred for result in results for pred in result]
    cache_predictions(train_predictions_cache_name, train_predictions)
```

```
def process_task(start, end, index, training_df, train_word_counts_pos, train_word_counts_neg, training_positive_prior, tr
    chunk = training_df.iloc[start:end]
    return bayesian_predictor(chunk, train_word_counts_pos, train_word_counts_neg,
                              training_positive_prior, training_negative_prior)
```

We can see in the output below, most examples have predictions which match the true sentiment.

```
################################################################################
                                    Task 5
################################################################################

Loaded data from cache: cache\movie_reviews-train-predictions-5-50.pkl

Training Data Predictions
                                                  Review Sentiment Prediction
1      It's a pretty good cast, but the film has nowh...  negative   negative
2      This ludicrous film offers the standard 1970's...  negative   negative
4      I really have to say, this was always a favori...  positive   positive
6      I love this movie. My friend Marcus and I were...  positive   positive
7      Street Fight is a brilliant piece of brutal sa...  positive   positive
...                                                  ...       ...        ...
49987  I am marking this as a "spoiler" only because ...  negative   negative
49988  After reading only two of the comments herein,...  positive   positive
49990  I remember seeing this years ago when it first...  positive   negative
49994  Whattt was with the sound? It sounded like it ...  negative   negative
49996  And maybe, as Fred Sandford used to say, "one ...  negative   negative

[25000 rows x 3 columns]
11705
13295
Training Data Accuracy: 0.83612
```

I also attempted to leverage my GPU using Cuda and Cupy to replace math library tasks such as math.log(), but I had difficulty with conflicts in my environment. The CPU pooling was sufficient to allow me to complete the task on time on this occasion.

This is something I will revisit for future assignments.

Task 6

By the nature of task 6, we have a lot of looping over previous work. The caching and thread pooling implemented in previous steps is hugely important in not repeating processing steps.

Task 6 loops through the entire project for values of minimum word length in the range of 1-10.

Using several arrays, I store the values of each loop, so I can refer to them by index.

This task also introduces the k-fold step, rather than splitting by value of the "split column". I use the library scikit learn, to run this process using their built-in function. This splits the data into 5 "folds" which are run through the functions of the previous tasks.

```
print_divider("-")
# print confusion matrix
print(f"Training Data Confusion Matrix")
for i in range(len(true_pos_array)):
    print(f"Minimum Word Length: {i + 1}")
    print(f"True Positive: {true_pos_array[i]}")
    print(f"True Negative: {true_neg_array[i]}")
    print(f"False Positive: {false_pos_array[i]}")
    print(f"False Negative: {false_neg_array[i]}")
    print_divider("-")

    # Percentage of True Positives, True Negatives, False Positives and False Negatives
    print(
        f"True Positive Percentage: {true_pos_array[i] / (true_pos_array[i] + false_pos_array[i]) * 100:.2f}%")
    print(
        f"True Negative Percentage: {true_neg_array[i] / (true_neg_array[i] + false_neg_array[i]) * 100:.2f}%")
    print(
        f"False Positive Percentage: {false_pos_array[i] / (true_pos_array[i] + false_pos_array[i]) * 100:.2f}%")
    print(
        f"False Negative Percentage: {false_neg_array[i] / (true_neg_array[i] + false_neg_array[i]) * 100:.2f}%")
    print_divider("-")
```

The code above prints several key pieces of information:

I print the confusion matrix for each iteration. Showing the number of correctly classified positives and negatives, alongside those which were incorrectly classified.

I then output the percentage for correct classifications by sentiment, and the percentages of incorrect classifications.

Lastly, The training score for each iteration, which is calculated by adding, the number of reviews which are correctly predicted, divided by the total reviews, for each fold. This is calculated for each iteration of minimum word size.

```
# Print the results
print_divider("-")
print(f"Average Data Accuracy: {sum(train_score_array) / len(train_score_array):.2f}")
print(f"Maximum Data Accuracy: {max(train_score_array):.2f}")
print(f"Maximum Data Score by Minimum Word Length: {train_score_array.index(max(train_score_array)) + 1}")
```

The result of my analysis is that the minimum word length which provides the best results is 3:

See the confusion matrix and true/false positive/negative results

```
--------------------------------------------------------------------------------

Minimum Word Length: 3
True Positive: 79298
True Negative: 88341
False Positive: 11659
False Negative: 20698

--------------------------------------------------------------------------------

True Positive Percentage: 87.18%
True Negative Percentage: 81.02%
False Positive Percentage: 12.82%
False Negative Percentage: 18.98%

--------------------------------------------------------------------------------
```

I also output the average and maximum accuracies. The best being length of 3 at 85%:

```
Average Data Accuracy: 0.84
Maximum Data Accuracy: 0.85
Maximum Data Score by Minimum Word Length: 3
```

Aside

I implemented some output formatting functions, to give visual indicators of progress, and make it clear which section of the code was being worked on.

A progress bar was added to areas with expensive loops, to show how far along the process is. An estimate of time remaining is calculated by recording the starting time, and interpolating time elapsed to the value it would be at 100%.

```
# Function to display a progress bar so the user knows the program is still running and how far along it is
def progressbar(i, upper_range, start_time):  2 usages   Emmett
    # Calculate the percentage of completion
    percentage = (i / (upper_range - 1)) * 100
    # Calculate the number of ▇ characters to display
    num_blocks = int(percentage)
    # Calculate elapsed time
    elapsed_time = time.time() - start_time
    # Estimate remaining time
    if percentage > 0:
        estimated_total_time = elapsed_time / (percentage / 100)
        remaining_time = estimated_total_time - elapsed_time
    else:
        remaining_time = 0
    # Create the progress bar string
    progress_string = f'\r{("▇" * num_blocks)}{("_" * (100 - num_blocks))} {percentage:.2f}% | Elapsed: {elapsed_time:.2f}s | Remaining: {remaining_time:.2f}s'
    if i == upper_range - 1:
        print(progress_string)
    else:
        print(progress_string, end='', flush=True)
```