Detection of Negative App Reviews Using Sentiment Analysis

DATA 245

Professor: Dr. Shih Yu Chang

Team 3:

Emma Hendry

Tanyu Yin

Yu Zhou

Yuanyuan Ruan

San Jose State University, CA

Table of Contents

1. Introduction

The purpose of this project is to develop and apply a methodology to detect negative Walmart App reviews retrieved from the Google Play Store. This project utilizes the Term Frequency- Inverse Document Frequency (TF-IDF) and word-embedding method to conduct document-based sentiment analysis to detect negative Walmart App reviews. Using the results of these methods, a best model is developed and tested on unseen reviews. The Non-negative Matrix Factorization (NMF) method and Latent Dirichlet Allocation (LDA) method are also used to extract topics from the reviews for use in aspect-based sentiment analysis. The long-term goal of this project is to develop a system for companies to detect negative reviews in real time and identify which aspects of their products are negatively reviewed.

2. Business Understanding

In the E-commerce era, consumers are increasingly willing to buy things and give feedback online. Customer product reviews is a key resource that is used to improve users' experience. How to respond to customer complaints timely and determine business values from consumer`s reviews is very important for business success. Our solution to the business problem described above is to develop an analysis system which detects real-time online negative reviews automatically and analyzes customers' sentiment changes over time. This system would require sentiment analysis from two perspectives, i.e., document-based and aspect-based. To lay the groundwork for this solution, this project focuses on conducting document-based sentiment analysis and aspect extraction for aspect-based sentiment analysis.

This project makes several assumptions. The first assumption is that star ratings are representative of the sentiment of the text review. Each review has a star rating between 1 and 5 where a score of 1 indicates a negative rating and a score of 5 indicates a positive rating. The star ratings were used to label the reviews as positive or negative, with ratings of 1, 2 and 3 labelled as negative and ratings of 4 and 5 labelled as positive. Another assumption is that although a rating of 3 could be considered "neutral" from a mathematical perspective, as it is in the middle of possible scores, from a business perspective this indicates a negative review.

After cleaning the datasets, we may apply several types of machine learning models, such as information-based, similarity-based, error-based, and probability-based models, and evaluate our models based on recall, precision and accuracy for classification.

Overall, we have used supervised machine learning methods to build binary classifiers and unsupervised machine learning methods to conduct aspect extraction.

3. Data Understanding

The dataset that was used for this project was scraped from Google play using the Google Play Scraper which provides APIs that can be used to retrieve data from the Google Play Store using Python (PlanB, n.d.). Using this scraper, 10,000 reviews for the Walmart App were scraped for dates between June 12th 2020 and October 14th 2020. The Walmart App was selected due to the relatively large proportion of negative reviews which suggests a business need for the detection of negative reviews (Figure 3.1).

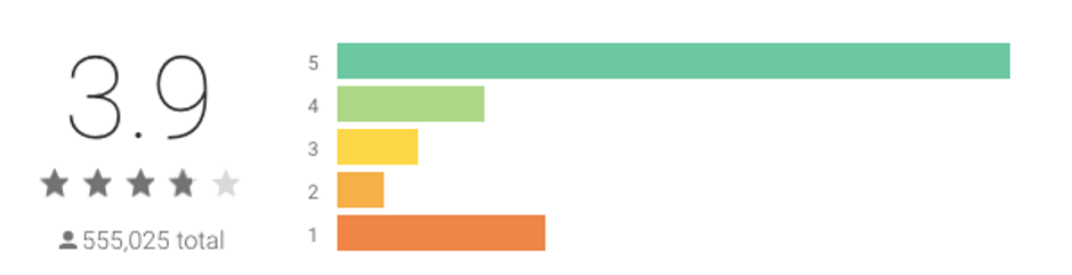

Figure 3.1: A screenshot showing the distribution of reviews on the Google Play Store for the Walmart App.

The scraped data consisted of 10 columns: the reviewId, username, userImage, content, score, thumbsUpCount, reviewCreatedVersion, at, replyContent, repliedAt (Figure 3.2). This project focuses on conducting document-based and aspect-based sentiment analysis and does not analyze temporal changes in reviews. Therefore, the relevant data fields for the analysis are the reviewId, which is used as a unique identifier for each review, the content field which is the raw text review, and the score which is used to create a positive or negative sentiment label.

	reviewId	userName	userImage	content	score	thumbsUpCount	reviewCreatedVersion	at	replyContent	repliedAt
0	gp:AOqpTOHuC9a634QjathsyDrU3ip9At5azd7X8o6RDzf...	galen johnson	https://lh3.googleusercontent.com/a-/AOh14GhyA...	Love Walmart and the Walmart app too. Lots of ...	5	2	20.38	2020-10-14 11:59:43	NaN	NaN
1	gp:AOqpTOF5XzVmyK52nMUPJO6R2RBU3SM241f5rO9PNTT...	Cedric Bernard	https://play-lh.googleusercontent.com/-2TNOOqB...	My new way of shopping	5	0	20.38	2020-10-14 11:54:11	NaN	NaN
2	gp:AOqpTOEhCsdm2GITL0qVcZdjVvPrziwp1isu_V2JmjM...	Michael Dawson	https://lh3.googleusercontent.com/a-/AOh14Gh5B...	I hated the approach to accessing each departm...	1	0	NaN	2020-10-14 11:51:46	NaN	NaN
3	gp:AOqpTOHUkzd3pDj9npbRNJQ0TLfTNyXeQr48DA0M_bV...	travelBandita	https://lh3.googleusercontent.com/a-/AOh14GhqR...	Please show where items are located in store w...	4	0	20.39	2020-10-14 11:46:02	NaN	NaN
4	gp:AOqpTOFtnMhygd5_WjznpeAQYN6ScNvzXwrCC1qgtYw...	Denise McFadden	https://lh3.googleusercontent.com/a-/AOh14GiAf...	Loving Walmart, Great deals,	5	0	20.39	2020-10-14 11:26:27	NaN	NaN

Figure 3.2: An overview of the raw data scraped from the Google Play Store

Although this dataset consists of usernames, to protect user privacy this field will not be used in the analysis and the processed review data will not be linked back to an individual user. Due to the small size of the dataset this dataset is stored locally. For a large-scale deployment, the anonymized data could be stored in the cloud.

In order to get to know our dataset better, we have written functions for data understanding. We have applied data quality techniques to check outliers (Figure 3.3) and missing values (Figure 3.4) in datasets. We have also checked the data types and non-null values for each feature using the built-in method "info" (Figure 3.5). In addition, we checked whether the rows in the dataset contains duplicates and found that there were no duplicated rows.

```
# check duplicated rows
data.duplicated().sum()
```

```
0
```

Figure 3.3: An output showing the duplicate records in raw data

```
# check missing values
from data_exploration import missing_values_check
missing_info = missing_values_check(reviews)
print(missing_info)
```

```
Unnamed: 0                0
reviewId                  0
userName                  1
userImage                 0
content                   0
score                     0
thumbsUpCount             0
reviewCreatedVersion    1921
at                        0
replyContent           9995
repliedAt              9995
norm_reviews             12
dtype: int64
```

Figure 3.4: An output showing the missing values in the raw data

```
reviews.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 0 to 9999
Data columns (total 12 columns):
Unnamed: 0              10000 non-null int64
reviewId                10000 non-null object
userName                9999 non-null object
userImage               10000 non-null object
content                 10000 non-null object
score                   10000 non-null int64
thumbsUpCount           10000 non-null int64
reviewCreatedVersion    8079 non-null object
at                      10000 non-null object
replyContent            5 non-null object
repliedAt               5 non-null object
norm_reviews            9988 non-null object
dtypes: int64(3), object(9)
memory usage: 1015.6+ KB
```

Figure 3.5: An output showing the characteristics of the raw data.

As we can see from Figure 3.6, we have review ratings ranging from 1 to 5, and most of the reviews have a score of 1, followed by a score of 5, with a small proportion of scores in the 2 to 4 star rating range. This shows that the reviews are polarized, indicating that some people did not like this application at all while others really liked it.

Similarly, we calculated the count of thumbsUp grouped by score (Figure 3.7). As shown, it has almost the same distribution as the score distribution, which also proved the polarization of the attitude toward this application. Due to the polarity of sentiment, there is a business need to discover why some users like the product while others do not, so improvements can be made.

```
[21]: import matplotlib.pyplot as plt

      plt.hist(raw_data['score'])
```

```
[21]: (array([4558.,    0.,  951.,    0.,    0.,  738.,    0.,  700.,    0.,
               3053.]),
        array([1. , 1.4, 1.8, 2.2, 2.6, 3. , 3.4, 3.8, 4.2, 4.6, 5. ]),
        <a list of 10 Patch objects>)
```

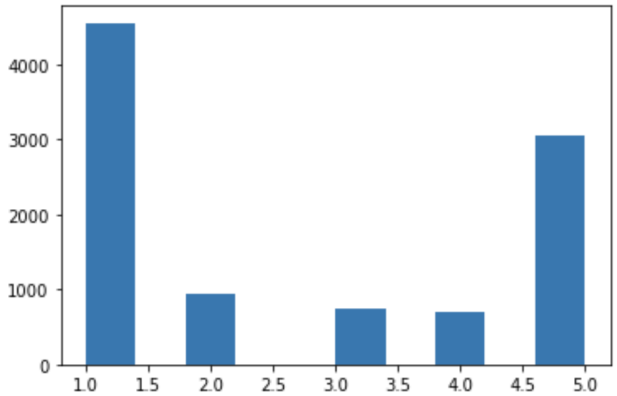

Figure 3.6: A bar graph showing the distribution of review ratings.

```
show_groupbythumbsUp_count(raw_data)
```

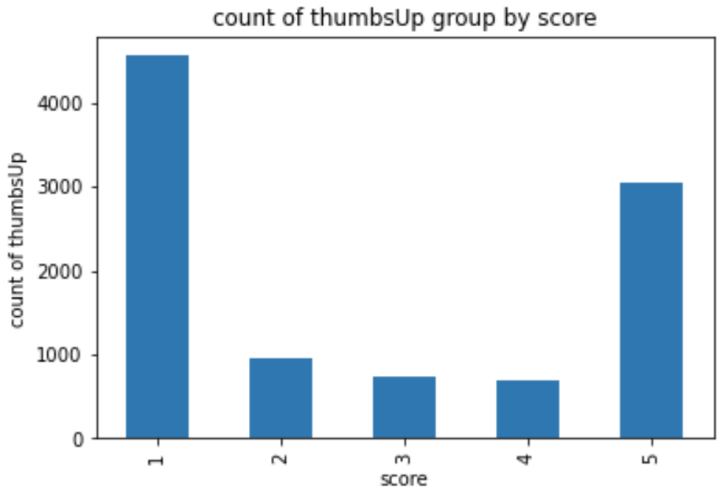

Figure 3.7: A bar graph showing the count of thumbsUp group by score

Shown as the histogram below, the review count changed over the month, the highest amount of reviews were written in July 2020 and the lowest happened in October (Figure 3.8). This may be caused by the high demand in shopping during summer break and the release of new versions.

```
raw_data = reformat_date(raw_data)
show_groupbymonth_count(raw_data)
```

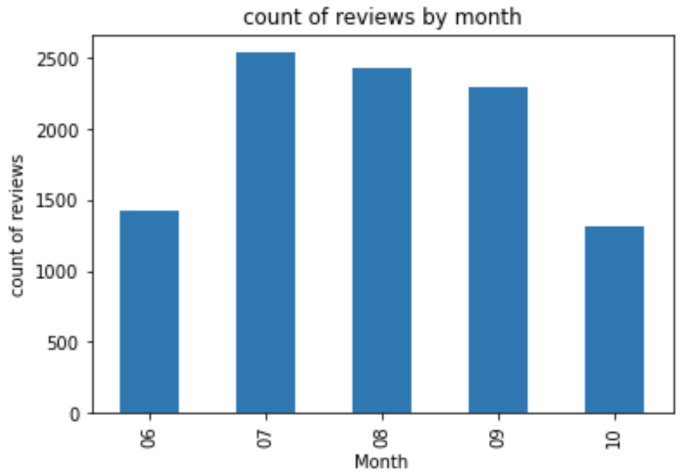

Figure 3.8: A bar graph showing the count of reviews by month

4. Data Preparation

4.1 Data Cleaning

4.1.1 Text Cleaning

After totally understanding what our raw data looks like, we started to prepare data for further modeling. Since the text part of raw reviews is our main data for analysis, we first focused on cleaning the text review data. We created a cleaning pipeline, consisting of 8 sub functions (see data_exploration.py). We first replaced emojis and removed all punctuations and extra whitespaces. Then we changed slangs and contractions into the normal format and removed digits. Lastly, we lemmatized the text, converted all words to lowercase, and removed stop words. Figure 4.1 shows this process.

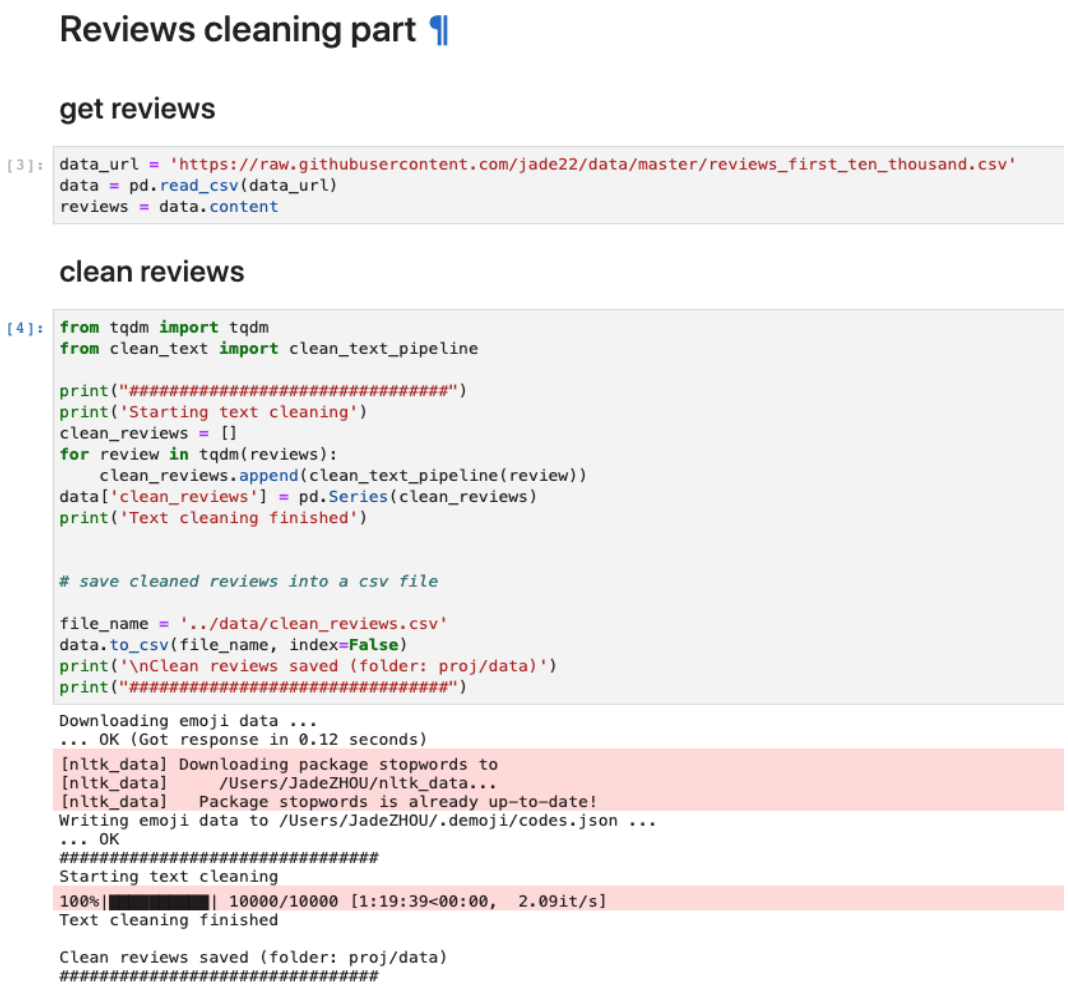

Figure 4.1: An output showing the reviews cleaning process.

Then we visualized the cleaned reviews to understand the most frequent words. From this we found that some common words, like "pron" and "app", were less informative but still in the reviews. These words were added to a list of common words and removed using a sub function called remove_common_words in the text cleaning pipeline.

After the above steps, we checked the cleaned text reviews again (Figure 4.2, 4.3). This showed that the cleaned text reviews were comparatively cleaner than the raw review data. The cleaned text data is used for the subsequent steps.

Figure 4.2: An output of word cloud

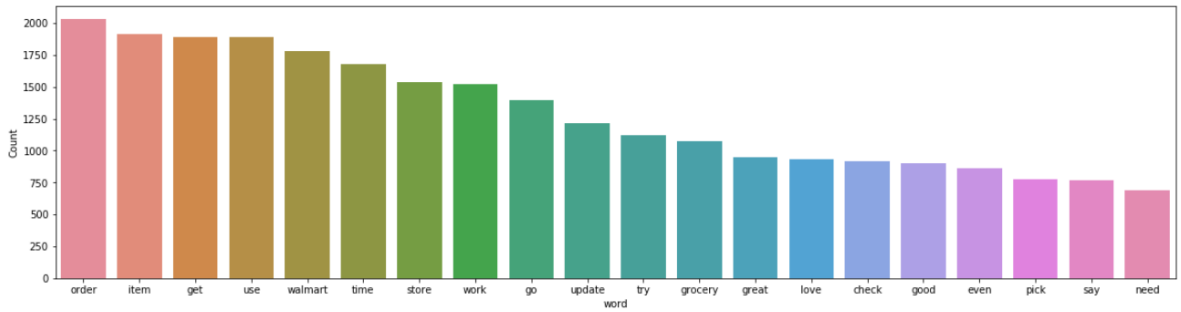

Figure 4.3: An output of the frequency of top words

Given that some reviews might change to null after the above text cleaning steps, we should check and handle them together with other columns in our dataframe in the next step.

4.1.2 Entire Dataset Cleaning

In this stage, we need to write some functions for data preparation. We utilized the missing_values_check function to check the whole dataset and found 18 missing values in the cleaned reviews. We directly deleted them with the handling_missing_values function. For the feature with more than 60% missing values, we removed that feature directly. The missing values of numeric features can be imputed using the mode. After removing missing values, we checked for duplicate rows and found no duplicate rows to be handled. If the dataset contained duplicated rows then the duplicated rows would have been dropped (Figure 4.4).

Whole dataset cleaning part

```
# read data

data = pd.read_csv('../data/clean_reviews.csv')
```

```
# check missing values
from dataset_preprocessing import missing_values_check
missing_values_check(data)
```

```
Unnamed: 0                0
Unnamed: 0.1              0
reviewId                  0
userName                  1
userImage                 0
content                   0
score                     0
thumbsUpCount             0
reviewCreatedVersion    1921
at                        0
replyContent           9995
repliedAt              9995
clean_reviews            18
dtype: int64
```

•••

```
# handle missing values
from dataset_preprocessing import handling_missing_values
missing_values_info, data = handling_missing_values(data)
print(missing_values_info)
```

```
reviewId                0
userName                0
userImage               0
content                 0
score                   0
thumbsUpCount           0
reviewCreatedVersion    0
at                      0
clean_reviews           0
dtype: int64
```

•••

```
# handle duplicates
from dataset_preprocessing import check_duplicates
data = check_duplicates(data)
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 9982 entries, 0 to 9999
Data columns (total 9 columns):
 #   Column                Non-Null Count  Dtype 
---  ------                --------------  ----- 
 0   reviewId              9982 non-null   object
 1   userName              9982 non-null   object
 2   userImage             9982 non-null   object
 3   content               9982 non-null   object
 4   score                 9982 non-null   int64 
 5   thumbsUpCount         9982 non-null   int64 
 6   reviewCreatedVersion  9982 non-null   object
 7   at                    9982 non-null   object
 8   clean_reviews         9982 non-null   object
dtypes: int64(2), object(7)
memory usage: 779.8+ KB
```

Figure 4.4: An output of whole dataset cleaning process

4.2 Data Transformations

To well prepare for feature engineering, we add a new feature called “review category” in which we transformed the scores in the range (1,3) into “negative” (labeled as 1 in the dataset as it is our target review for analyzing) and transformed the scores 4 and 5 into “positive” (labeled as 0 in the dataset), we can see that there are 6240 negative and 3742 positive reviews in the raw dataset (Figure 4.6).

Add review category (positive = 4 or 5, negative = 1 - 3 score)

```
raw_data['review_category'] = [0 if x == 5 or x == 4 else 1 for x in raw_data['score']]
raw_data.review_category.value_counts()
```

```
1    6240
0    3742
Name: review_category, dtype: int64
```

Figure 4.6: Function to add review category

In addition, we extracted the year, month, day, hour and minute from the column “at” which is the date and time of each review (Figure 4.7). So we can do more analysis with different time dimensions.

```
def reformat_date(data):
    split = pd.to_datetime(data['at'], format='%Y-%m-%d %H:%M:%S')
    data['Year'] = split.dt.strftime('%Y')
    data['Month'] = split.dt.strftime('%m')
    data['Day'] = split.dt.strftime('%d')
    data['Hour'] = split.dt.strftime('%H')
    data['Minute'] = split.dt.strftime('%M')
    return data
```

Figure 4.7: Function to extract the year, month, day, hour and minute from the "at" field.

4.3 Data Scaling

Using the normalize_data function (Figure 4.8), the thumbsUpCount feature was normalized to a range of (0,1).

```
[18]: # normalize date
      from dataset_preprocessing import normalize_data
      data = normalize_data(data, 'thumbsUpCount')
```

	score	thumbsUpCount
count	9982.000000	9964.000000
mean	2.672210	0.004259
std	1.761147	0.030728
min	1.000000	0.000000
25%	1.000000	0.000000
50%	2.000000	0.000000
75%	5.000000	0.001000
max	5.000000	1.000000

Figure 4.8: An output of normalization for thumbsUpCount feature

4.4 Data Selection

After data transformation and normalization, we got 16 columns in total (Figure 4.9). For our modeling, we mainly focused on the sentiment analysis (review category prediction) and aspect extraction. Thus we select the reviewId, content and review category as selected features in our model.

```
data_cleaned.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9982 entries, 0 to 9981
Data columns (total 16 columns):
Unnamed: 0              9982 non-null int64
reviewId                9982 non-null object
userName                9982 non-null object
userImage               9982 non-null object
content                 9982 non-null object
review_score            9982 non-null int64
thumbsUpCount           9964 non-null float64
reviewCreatedVersion    9982 non-null object
at                      9982 non-null object
Year                    9982 non-null int64
Month                   9982 non-null int64
Day                     9982 non-null int64
Hour                    9982 non-null int64
Minute                  9982 non-null int64
review_category         9982 non-null int64
clean_reviews           9982 non-null object
dtypes: float64(1), int64(8), object(7)
memory usage: 1.2+ MB
```

Figure 4.9: Information about the cleaned whole dataset.

5. Feature Engineering

5.1 TF-IDF

The first method that was used to extract features from the cleaned text reviews is the Term Frequency – Inverse Document Frequency (TD-IDF) method. This method vectorizes the text so it can be interpreted by the machine learning algorithm and computes weights for each weight indicating the importance of the word in the document (Scott, 2019). For this project, the "term" refers to the words that are found in the reviews and the "document" refers to the reviews.

The first component of TF-IDF is the term frequency, which is a measure of the frequency of words in the document (Scott, 2019). The term frequency counts all the words in the document and normalizes the values. For this project the Euclidean method of normalization was used. The equation for the calculation of term frequency is shown below.

$$TF = \frac{count\ of\ \ terms\ in\ document}{number\ of\ \ words\ in\ document}$$

In these counts of words there will be a few words that are very common in the document and a lot of words that occur rarely in the document. This does not necessarily reflect the importance of the word in the document, so another calculation is required (Scott, 2019). Document frequency is the number of documents (in our case, reviews) where each term occurs. This value is normalized by dividing the document frequency by the total number of documents (reviews).

$$DF = occurence\ of\ \ terms\ in\ document$$

To assess the importance of a particular term, the inverse document frequency is calculated. This is calculated by dividing the count of all the words in the corpus (all the terms found in all reviews) by the document frequency. To smooth values which would have a document frequency value of 0, a value of 1 is added to the calculation as shown below. Additionally, to limit the inverse document frequency values from getting too large with large corpora, the log of the equation is taken (Scott, 2019).

$$IDF = \log\left(\frac{N}{DF+1}\right)$$

Lastly, the final TF-IDF value is calculated by multiplying the term frequency and inverse document frequency together as shown below (Scott, 2019).

$$TF\text{-}IDF = TF \cdot \log\left(\frac{N}{DF+1}\right)$$

Before implementing the TF-IDF method, the prediction problem is converted to a binary classification problem by converting all reviews with a score of 1, 2 or 3 to a negative review (encoded as 1) and all reviews with a score of 4 or 5 to a positive review (encoded as 0). The dataset was then split into testing and training data and the training data was down-sampled to balance the dataset between the positive and negative reviews.

As shown by the function below (Figure 5.1), the TF-IDF method was implemented in Python using the TfidfVectorizer. As shown by the function below, the TfidfVectorizer is first fitted to the training dataset to create the training TF-IDF matrix (Figure 5.2). To create the testing TF-IDF matrix, the vectorizer is transformed but not fitted to the test dataset. This ensures that the features are extracted from the training data, and that both the training and testing data contain identical features. A second function was used to

apply the function below to the data frames containing the split data. The TF-IDF method was tested using unigrams and bigrams and unigrams produced the best results in terms of accuracy, so unigrams were used to generate the TF-IDF matrix.

```
def tf_idf_weight_norm_extract(train_data, test_data, raw_data_col, norm = "l2", max_features=None,
                               max_df = 1.0, min_df = 1, ngram_range=(1,1)) :
    '''
    Calculate normalized tf-idf weight.
    @data: the dataframe containing the raw (unvectorized) data training.
    Please note that this is a modified version of the tf-idf function
    tf_idf_weight_norm. we are using the TfidfVectorizer module so the input is the unvectorized data.
    @raw_data_col : the column containing the unvectorized cleaned reviews. e.g. clean_reviews
    @norm: a string ('l2', 'l1'), representing normalization method
    @max features : only consider the top max_features ordered by term frequency across the corpus
    @max_df : ignore terms that have a document frequency strictly higher than the given threshold
    @min_df : ignore terms that have a document frequency strictly lower than the given threshold
    @df_type : indicates if the tf-idf is being generated for 'train' or 'test' data
    @return: a dataframe
    '''

    data_array = list(train_data[raw_data_col])
    data_array_test = list(test_data[raw_data_col])

    if norm == 'l2': # Euclidean normalization
        tf_idf = TfidfVectorizer(max_features = max_features, max_df = max_df, min_df = min_df,
                                 norm = 'l2', token_pattern=r"\b\w+\b", stop_words=None,
                                 ngram_range=ngram_range, analyzer='word')
        tf_idf_weight_norm_train = tf_idf.fit_transform(data_array).todense()
        tf_idf_weight_norm_test = tf_idf.transform(data_array_test).todense()

    elif norm == 'l1': # Manhattan normalization
        tf_idf = TfidfVectorizer(max_features = max_features, max_df = max_df, min_df = min_df,
                                 norm = 'l1', token_pattern=r"\b\w+\b", stop_words=None,
                                 ngram_range=ngram_range, analyzer='word')
        tf_idf_weight_norm_train = tf_idf.fit_transform(data_array).todense()
        tf_idf_weight_norm_test = tf_idf.transform(data_array_test).todense()

    else:
        print("method must be 'l1 or 'l2', default is l2")

    tf_idf_weight_norm_train = pd.DataFrame(tf_idf_weight_norm_train,
                                            columns = tf_idf.get_feature_names())
    tf_idf_weight_norm_test = pd.DataFrame(tf_idf_weight_norm_test,
                                           columns = tf_idf.get_feature_names())

    return tf_idf_weight_norm_train, tf_idf_weight_norm_test
```

Figure 5.1: Function to generate the TF-IDF data for the test and train dataset

```
train_tfidf.iloc[0:20,0:10]
```

reviewId	a	ab	abandon	abide	ability	abit	able	abou	absolute	absolutely
gp:AOqpTOF7fGl3wWeaL639DwhqLYblowkZxYttf8H6tD738G-ZsfObyUpB_vw2S6ruEYe0Ju36xeQMwh7lmb0aPOY	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOHsKHTu5Bw9DlQWtijC6Ut7_G7zzk4MeqUyYHYGQjPgVuB2qpsTYMsM_RV-iLRuqW9ETQcAlcDd0ySLvGI	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOFNDlYb1ECA_YP_tffKliC8WBjvJfpoPyLFi-1co9KeJn4OUG8YPOgrUAXVVfBquzIvsSSeL9OOWMjkogI	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOEp-8hoE_kqGBuu5haTE7z_9Pw6SZQh5HyqOhh5zdl64elc2uk5WWT1-TXxgQqgvRluaXcDJdNANs-qNQc	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOH-USDscUgE9zpv_G1u1A_xchgrx5GAk8L7XAwN9e4fI-nre-wdo-wdWLzyQFX0AMvGa8uvo6Vjt_DOigw	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOF7JlmcDUaS3IRB9FARlNoe8SoDb4wTh9NnR1GjRgovJdJUYb_bAfe9QCukt8E40GFuzamBk7VeyYg5riQ	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOFo35K9NGWb_nToNl6QArHY2x-ey0rO7C5lm08tele84eK3avwU__IJluQZmP5lm0ljuWXZ8TEXppyX5gc	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOHXM9VbJARvlGH-WCqlpwKkpRGUKsKSAXfw5uczMP8tg2UcBHX4J1_bVbT0yX1TMwsl7QtSXp3qkeoTBKY	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOHWm9k03FQfo5aeBOKczuK5KFo24ExOKkTUd0LI_B58xn3diaK0Nc-esWkYBa6QBm9YQK2BlObPNrWvIQQ	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOHy3FDZN12UEVpfCPRlyOJ4bkDramz_kSj2Cn4mxq2dPAjN10lktho87n9u6Ut6fjSLohGeRPJCVhjAkH4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.680092
gp:AOqpTOEEQQrXuLH_hjEKmDrAyhRw82Sw-QNvreGGvJG_WH3guhVip-iLY9Ea_qYPoSbwH-_io8HsEeb0DAetXlc	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.812251
gp:AOqpTOEk3tgZaJZFslMzFovz0ZTf-hW6czBTO2o8rrrkdoz1kDnLcXtPD6XcXHdLpwQ6j9nMlNefGu2Mvj4T0H0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOG6fWeTLDa6EQ7RDnzpIXMmCVTFGbUqM23u3xoXnZDeyqvaWRFBFyLlmZoBI0OEttnt0B7wAP5ep7WR1iM	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOGIjTE0gnilA3m3pz2zYGITx8SLMbvE124leBdlGrPOKYKVs9B5bupTYS-ICN54LKHselUBipuyDbozRGY	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOH31-irtyFpUdXsuLZPdnDhZ2SP7taGtT4D5fxbbMlYR3oNjEOQmRQT0dgRdE6RUv4AkWfpONk39CmpJT8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOF9cSSSsiLQOPHXJUUg8D2lvfGmz8ytex3vXJ9teOHUYcho0iBesHeUMY-nv3q3u3sbW2Pr6v5JXe-5imE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOH5XsGo7c7QVzqSHIFbvlcrWtJdLHnv0Pz8cr1koOelNd5zjNg545oQL5OkBokN8R3pjfuBiAg8QBrZhjE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOGRk0UNzFa_U6-hJsZm4ygODHik7C_jmncHhQ43wRorUn49Cltbbv5vLzhsJzRMwKjwO8s1R5VEfRqOdbE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOGcaeNNSi8q1i8iYHfe9tgbqo607zWilkHw8EJlEEEAzA6XRcYKxfxa30QCe-B2bxtCF4GiUZ0WyFkvS_Q	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000
gp:AOqpTOEXRAD4ShHXHZyb6epgl4UjAD3GdZRAladKyXZvuA5nS4hHwlphClVhzl1S2LI7-GcbHL2tQ0wdtaLOybE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.000000

Figure 5.2: A screenshot of the TF-IDF matrix for the training set

5.2 Word Embedding

Given that features generated via TF-IDF are only able to represent the word frequency but ignore the relationship between different words, its result may be not accurate enough. So we also tried another more advanced method, that is, word embedding. Its main idea is mapping a word by a vector space of real values, which can express the semantic relationship of two words. In our project, we employed a pre-trained dictionary named wiki-news-300d-1M.vec (Mikolov et al., 2017), which comes from fastText library. This dictionary was trained on three sources, including Wikipedia 2017, UMBC web-based corpus and statmt.org news dataset. It contains one million unique words and each word is represented by 300 features. Based on this, we mapped our reviews into 300 dimensions for the next modeling part. The Figure 5.4 and Figure 5.5 show our final features generated via word embedding.

```
# load the pre-trained word embedding dictionary

from reviews_embedding import load_word_embeds

fname_word_embeds = 'wiki-news-300d-1M.vec' # first download this file from https://fasttext.cc/docs/en/english-vectors.html
word_embeds = load_word_embeds(fname_word_embeds)
```

```
975it [00:00, 9746.22it/s]

##############################
Loading pretrained word embedding dictionary:
999995it [03:56, 4234.04it/s]
Got pretrained word embedding dictionary.
```

```
# map reviews via word embedding

from reviews_embedding import reviews_embedding_and_save

X_train_fname = '../data/X_train.csv' # file to save train features
X_test_fname = '../data/X_test.csv' # file to save test features
reviews_embedding_and_save(reviews_train, word_embeds, X_train_fname)
reviews_embedding_and_save(reviews_test, word_embeds, X_test_fname)
```

```
  1%|          | 45/5134 [00:00<00:13, 376.99it/s]

Starting reviews embedding:
100%|██████████| 5134/5134 [00:23<00:00, 218.31it/s]
Shape of embed_features: (5134, 300)
  5%|▌         | 149/2995 [00:00<00:01, 1487.84it/s]

Reviews embedding saved in a .csv file.
##############################

Starting reviews embedding:
100%|██████████| 2995/2995 [00:08<00:00, 335.37it/s]
Shape of embed_features: (2995, 300)

Reviews embedding saved in a .csv file.
##############################
```

Figure 5.4: Generating features via word embedding

	0	1	2	3	4	5	6	7	8	9	...	290	291	292	293	294	295	296	297	298	299
0	-0.065800	-0.005625	-0.042475	-0.050050	-0.072050	0.063675	-0.016750	0.064450	-0.036675	0.028750	...	-0.018250	-0.036625	-0.035125	0.020000	0.036025	0.138425	-0.007575	0.093975	-0.042975	0.127625
1	-0.090350	-0.012050	-0.106750	0.031600	-0.005825	-0.023550	-0.008950	-0.028500	-0.020075	0.021175	...	-0.014750	-0.019825	-0.030775	0.054900	0.014975	-0.015400	-0.093550	0.206175	0.072750	0.001775
2	-0.158800	-0.069100	-0.087200	-0.017400	0.021800	-0.027600	-0.013300	0.022200	0.028500	0.034100	...	0.007800	0.059400	-0.121400	0.021000	0.139800	-0.081500	-0.118200	0.261700	-0.024700	-0.026100
3	0.017300	-0.107925	-0.089275	-0.021550	0.052975	-0.045350	-0.001225	-0.025000	0.027600	0.001500	...	0.066225	0.043150	-0.000900	0.089275	0.091750	-0.020700	-0.047275	0.072775	-0.068600	-0.051175
4	-0.024657	0.013366	-0.023430	-0.005761	0.004309	-0.024861	-0.032182	-0.005827	-0.024720	-0.029186	...	0.001339	0.000575	-0.009927	0.006357	0.006198	-0.014477	-0.039086	0.134341	0.027870	0.000180
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
5129	-0.029233	0.016176	-0.007157	-0.000295	0.008148	-0.061157	-0.019171	0.001748	-0.013562	-0.033214	...	-0.003176	-0.019214	-0.015852	0.031929	0.045586	-0.007376	-0.031562	0.144224	0.020167	0.008395
5130	-0.015100	0.012783	0.012500	-0.027617	-0.055000	0.016650	-0.044600	0.022217	-0.017250	0.004550	...	-0.027717	-0.028833	-0.043167	0.012233	-0.011833	-0.013483	-0.043783	0.147883	0.011983	0.017917
5131	-0.058900	-0.034200	-0.048675	-0.039475	0.026075	0.032825	-0.021575	-0.041425	0.063450	-0.038975	...	0.052925	-0.008425	-0.044850	0.036625	-0.014325	0.039575	-0.091050	0.162225	-0.064175	0.009100
5132	0.014161	-0.000911	-0.026467	-0.002022	-0.031706	-0.056706	-0.005817	0.017261	-0.012111	0.010728	...	-0.006378	-0.001400	-0.034944	0.007206	0.067883	-0.003044	-0.037672	0.144144	0.024211	0.016872
5133	-0.158250	-0.003550	-0.191150	0.061300	-0.040200	0.058900	0.068150	0.086450	0.057250	-0.090000	...	0.047950	0.094050	-0.051350	0.010850	-0.037400	0.200100	0.010850	0.254850	-0.178600	0.017950

5134 rows × 300 columns

Figure 5.5: A screenshot of the reviews embedding for the training set

6. Modelling & Evaluation

6.1 Document-based Sentiment Analysis

6.1.1 Features Generated Using TD-IDF

6.1.1.1 Baseline Models: Decision Tree, Random Forest, Multinomial Naïve Bayes, SGD Classifier

Once the test and train features had been generated using the TF-IDF method, baseline models were created using a Decision Tree, Random Forest, SGDClassifier and Multinomial Naïve Bayes model (Figure 6.1- 6.4). The models results are compared in more detail in Table 6.1.

```
dct = apply_model(model_type='dt',tfidf_train = train_tfidf, y_train_ = y_train_s,
                  tfidf_test = test_tfidf, y_test_ = y_test, method = 'gini')
```

```
Accuracy score: 81.703%

Detailed classification report:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support

           0       0.74      0.82      0.78      1175
           1       0.88      0.81      0.84      1820

    accuracy                           0.82      2995
   macro avg       0.81      0.82      0.81      2995
weighted avg       0.82      0.82      0.82      2995
```

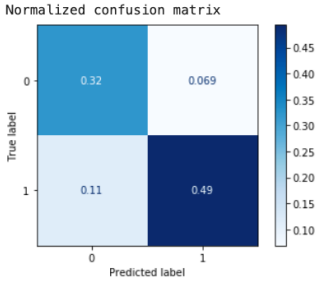

Figure 6.1: The results of the baseline Decision Tree model.

```
rf = apply_model(model_type='rf',tfidf_train = train_tfidf, y_train_ = y_train_s,
                 tfidf_test = test_tfidf, y_test_ = y_test, method = 'gini')
```

```
Accuracy score: 86.678%

Detailed classification report:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support

           0       0.83      0.83      0.83      1175
           1       0.89      0.89      0.89      1820

    accuracy                           0.87      2995
   macro avg       0.86      0.86      0.86      2995
weighted avg       0.87      0.87      0.87      2995
```

Normalized confusion matrix

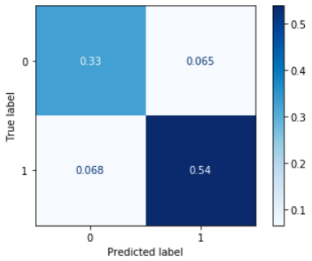

Figure 6.2: The results of the baseline Random Forest model.

```
mnb = apply_model(model_type='mnb',tfidf_train = train_tfidf, y_train_ = y_train_s,
                  tfidf_test = test_tfidf, y_test_ = y_test, method = 'gini')
```

```
Accuracy score: 88.447%

Detailed classification report:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support

           0       0.89      0.81      0.85      1175
           1       0.88      0.93      0.91      1820

    accuracy                           0.88      2995
   macro avg       0.88      0.87      0.88      2995
weighted avg       0.88      0.88      0.88      2995
```

Normalized confusion matrix

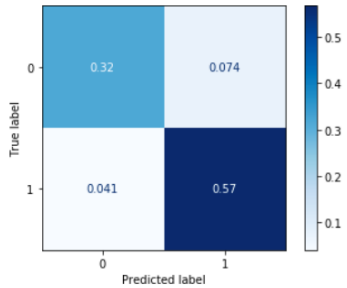

Figure 6.3: The results of the baseline Naïve Bayes model.

```
Accuracy score: 88.648%

Detailed classification report:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support

           0       0.86      0.85      0.85      1175
           1       0.90      0.91      0.91      1820

    accuracy                           0.89      2995
   macro avg       0.88      0.88      0.88      2995
weighted avg       0.89      0.89      0.89      2995
```

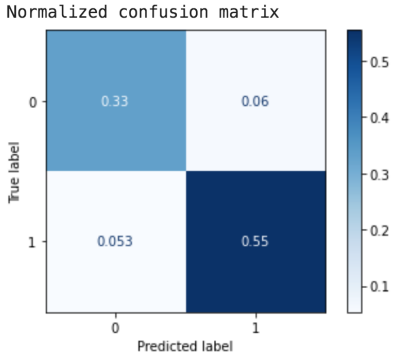

Figure 6.4: The results of the baseline SGDClassifier model.

Table 6.1: The results of the baseline models

Classifier Name	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Decision Tree	81.70	88.00	81.00	84.00
Random Forest	86.68	89.00	89.00	89.00
Multinomial Naïve Bayes	88.45	88.00	93.00	91.00
SGDClassifier	88.65	90.00	91.00	91.00

Note: The Precision, Recall and F1-Score shown are related to the classification of negative reviews (label = 1).

As shown by table 6.1, the SGDClassifier performs best in terms of overall accuracy and precision and the Multinomial Naïve Bayes performs best in terms of recall. Both Multinomial Naive Bayes and SGDClassifier perform equally well in terms of the F1-score. The recall rate is especially relevant for this business case, as we want to select the model that is the best at extracting negative reviews. The recall rate for Multinomial Naïve Bayes is the highest which means that it is the most sensitive to negative reviews. The recall rate for the Multinomial Naïve Bayes classifier indicates that there is a 93% chance that the model will correctly detect negative reviews.

6.1.1.2 Parameter Tuning: Decision Tree, Random Forest, Multinomial Naïve Bayes and SGD Classifier

Next, to try and improve upon the results parameter tuning was conducted using GridSearchCV for Multinomial Naïve Bayes and RandomizedSearchCV for the Decision Tree and Random Forest Model. A randomized method was used for the Decision Tree and Random Forest Model due to the large number of possible parameter combinations being tested. The tuned Decision Tree Model had a slightly lower overall accuracy of 78.90% compared with 81.70% for the baseline model. The tuned Random Forest Model had a slightly lower overall accuracy of 86.54% compared with 86.68% for the baseline model. The tuned Multinomial Naïve Bayes model had an accuracy of 88.45% which was the same as the baseline model. The tuned SGDClassifier model had an accuracy of 88.65% which was the same as the baseline model. Based on the results of this parameter tuning, the baseline models were used for the subsequent steps.

6.1.1.3 K-Fold Cross Validation: Decision Tree, Random Forest, Multinomial Naïve Bayes and SGDClassifier

Next, to more accurately assess the performance of the classifiers K-Fold Cross Validation was conducted on 5 different folds of the data. Within each fold of the data, down-sampling and TF-IDF vectorization were conducted on the training dataset to ensure that the pre-processing methods were kept consistent.

For each of the folds, the overall accuracy score was outputted as well as a detailed classification report and normalized confusion matrix (Figure 6.5). Additionally, the average accuracy score was also calculated for each model. Table 6.2 compares the average accuracy and the average precision, recall and F1-score for negative reviews for the classifiers. As shown, the Multinomial Naïve Bayes classifier performed best in terms of overall accuracy, recall and F1-score. The SGDClassifier had the best results in terms of precision. As the recall rate was the highest for the Multinomial Naïve Bayes classifier, this is considered the best model when using the TF-IDF method based on the collected data.

```
kFoldCrossVal(5, dct, x_train=X_train, y_train=y_train)
Evaluation Metrics for Fold=1
Accuracy score: 79.399%

Detailed classification report:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support

           0       0.69      0.81      0.75       521
           1       0.88      0.78      0.83       877

    accuracy                           0.79      1398
   macro avg       0.78      0.80      0.79      1398
weighted avg       0.81      0.79      0.80      1398
```

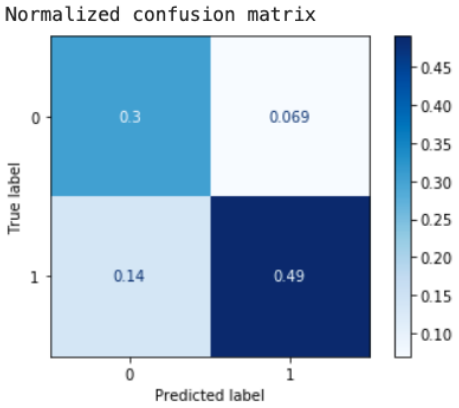

Figure 6.5: An example of the evaluation metrics that were outputted for each fold during K-Fold Cross Validation.

Table 6.2: The results of cross validation

Classifier Name	Accuracy (%)	Precision (%)	Recall (%)	F1-Score (%)
Decision Tree	80.61	88.20	80.40	84.00
Random Forest	86.12	88.80	89.60	89.20
Multinomial Naïve Bayes	89.07	90.20	92.80	91.60
SGDClassifier	88.91	91.60	90.80	91.40

Note: The Precision, Recall and F1-Score shown are related to the classification of negative reviews (label = 1).

6.1.2 Features Generated Using Word Embedding

6.1.2.1 Baseline Models

Using features generated by word embedding, we considered 7 classifiers, consisting of 5 basic ML classifiers (Decision Tree, KNN, GNB, Logistic Regression, SVM) and 2 ensembling classifiers (Random Forest, Gradient Boosting). Then we found the top 5 according to the result of cross-validation and treated them as the baseline models. Figure 6.6 shows that among the top 5, gradient boosting and random forest are overfitting. So we excluded them and the baseline models are Logistic Regression, SVM, and KNN.

```
# get baseline models

from modeling_and_evaluation import compare_clfs_perf

random_state = 0
cv = StratifiedKFold(n_splits=10, shuffle=True)
clfs = [DecisionTreeClassifier(min_samples_split=10, random_state=random_state),
        KNeighborsClassifier(),
        GaussianNB(),
        LogisticRegression(random_state = random_state,),
        SVC(random_state=random_state),
        RandomForestClassifier(random_state=random_state),
        GradientBoostingClassifier(random_state=random_state)]

compare_clfs_perf(clfs, X_train, y_train, X_test, y_test, cv, scoring='accuracy', n_jobs=-1)
```

```
100%|██████████| 7/7 [04:52<00:00, 41.76s/it]
```

	clf name	clf params	clf validation accuracy std (CrossVal)	clf train accuracy mean (CrossVal)	clf validation accuracy mean (CrossVal)	accuracy (testing dataset)	clf time
0	SVC	{'C': 1.0, 'break_ties': False, 'cache_size': ...	0.0180457	0.906657	0.878459	0.89015	4.25944
1	LogisticRegression	{'C': 1.0, 'class_weight': None, 'dual': False...	0.0135178	0.88456	0.871832	0.887813	0.138252
2	GradientBoostingClassifier	{'ccp_alpha': 0.0, 'criterion': 'friedman_mse'...	0.00894646	0.936458	0.862679	0.87813	48.4714
3	RandomForestClassifier	{'bootstrap': True, 'ccp_alpha': 0.0, 'class_w...	0.0116557	0.990802	0.850213	0.86778	7.12819
4	KNeighborsClassifier	{'algorithm': 'auto', 'leaf_size': 30, 'metric...	0.0139174	0.872397	0.829965	0.855426	0.0998206
5	DecisionTreeClassifier	{'ccp_alpha': 0.0, 'class_weight': None, 'crit...	0.0136176	0.971021	0.765681	0.758932	1.85357
6	GaussianNB	{'priors': None, 'var_smoothing': 1e-09}	0.0169525	0.7301	0.730035	0.770952	0.0401606

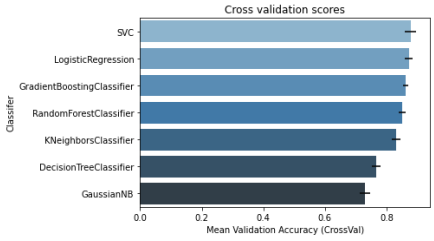

Figure 6.6: Performance comparison of models via cross validation

6.1.2.2. Hyper-parameters Tuning: KNN, Logistic Regression, SVM

Next, we got the best parameter setting for each classifier after tuning the baseline models (Figure 6.7-6.9). Then we cross validated them again to make sure there's no overfitting issue in the tuned models (Figure 6.10).

KNN

```
from modeling_and_evaluation import tune_params
cv = StratifiedKFold(n_splits=10,shuffle=True)
random_state = 0
knn = KNeighborsClassifier(n_jobs=-1)
knn_p_grid = {'n_neighbors': range(1,15,1)}
knn_best = tune_params(clf=knn, p_grid=knn_p_grid, cv=cv, X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test)
```

```
Best parameters set found on development set:

{'n_neighbors': 6}

Grid scores on development set:

0.809 (+/-0.021) for {'n_neighbors': 1}
0.811 (+/-0.017) for {'n_neighbors': 2}
0.831 (+/-0.019) for {'n_neighbors': 3}
0.838 (+/-0.016) for {'n_neighbors': 4}
0.829 (+/-0.020) for {'n_neighbors': 5}
0.840 (+/-0.017) for {'n_neighbors': 6}
0.831 (+/-0.019) for {'n_neighbors': 7}
0.836 (+/-0.017) for {'n_neighbors': 8}
0.832 (+/-0.019) for {'n_neighbors': 9}
0.835 (+/-0.019) for {'n_neighbors': 10}
0.828 (+/-0.021) for {'n_neighbors': 11}
0.834 (+/-0.021) for {'n_neighbors': 12}
0.824 (+/-0.021) for {'n_neighbors': 13}
0.830 (+/-0.019) for {'n_neighbors': 14}

Detailed classification report:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support

           0       0.85      0.78      0.81      1175
           1       0.86      0.91      0.88      1820

    accuracy                           0.86      2995
   macro avg       0.85      0.84      0.85      2995
weighted avg       0.86      0.86      0.85      2995
```

Normalized confusion matrix

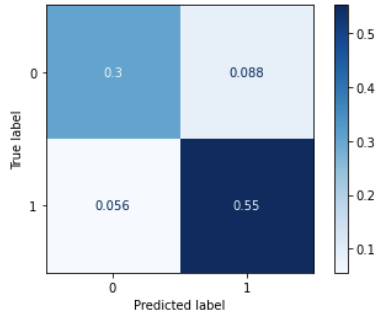

Figure 6.7: Best parameter setting of KNN

Logistic Regression

```
from modeling_and_evaluation import tune_params
cv = StratifiedKFold(n_splits=10,shuffle=True)
random_state = 0
lr = LogisticRegression(random_state=random_state, max_iter=500, n_jobs=-1)
lr_p_grid = dict(C=[1,2,4,10,20,40,80,160])
lr_best = tune_params(clf=lr, p_grid=lr_p_grid, cv=cv, X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test)
```

```
Best parameters set found on development set:

{'C': 4}

Grid scores on development set:

0.867 (+/-0.011) for {'C': 1}
0.872 (+/-0.014) for {'C': 2}
0.872 (+/-0.014) for {'C': 4}
0.872 (+/-0.012) for {'C': 10}
0.871 (+/-0.014) for {'C': 20}
0.867 (+/-0.014) for {'C': 40}
0.865 (+/-0.017) for {'C': 80}
0.863 (+/-0.017) for {'C': 160}

Detailed classification report:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support

           0       0.87      0.84      0.85      1175
           1       0.90      0.92      0.91      1820

    accuracy                           0.89      2995
   macro avg       0.88      0.88      0.88      2995
weighted avg       0.89      0.89      0.89      2995
```

Normalized confusion matrix

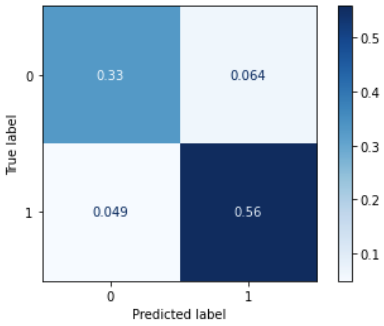

Figure 6.8: Best parameter setting of Logistic Regression

SVC

```
from modeling_and_evaluation import tune_params
cv = StratifiedKFold(n_splits=10, shuffle=True)
random_state = 0
svc = SVC(random_state=random_state)
svc_p_grid = [{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
              {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']}]
svc_best = tune_params(clf=svc, p_grid=svc_p_grid, cv=cv, X_train=X_train, y_train=y_train, X_test=X_test, y_test=y_test)
```

```
Best parameters set found on development set:

{'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}

Grid scores on development set:

0.875 (+/-0.012) for {'C': 1, 'kernel': 'linear'}
0.867 (+/-0.011) for {'C': 10, 'kernel': 'linear'}
0.865 (+/-0.012) for {'C': 100, 'kernel': 'linear'}
0.865 (+/-0.012) for {'C': 1000, 'kernel': 'linear'}
0.687 (+/-0.013) for {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'}
0.596 (+/-0.119) for {'C': 1, 'gamma': 0.0001, 'kernel': 'rbf'}
0.812 (+/-0.018) for {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
0.687 (+/-0.013) for {'C': 10, 'gamma': 0.0001, 'kernel': 'rbf'}
0.866 (+/-0.011) for {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'}
0.812 (+/-0.017) for {'C': 100, 'gamma': 0.0001, 'kernel': 'rbf'}
0.875 (+/-0.011) for {'C': 1000, 'gamma': 0.001, 'kernel': 'rbf'}
0.867 (+/-0.012) for {'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}

Detailed classification report:

The model is trained on the full train set.
The scores are computed on the full test set.

              precision    recall  f1-score   support

           0       0.87      0.84      0.85      1175
           1       0.90      0.92      0.91      1820

    accuracy                           0.89      2995
   macro avg       0.88      0.88      0.88      2995
weighted avg       0.89      0.89      0.89      2995
```

```
Normalized confusion matrix
```

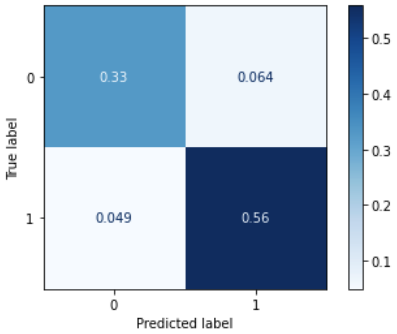

Figure 6.9: Best parameter setting of SVC

```
# cross validate the tuned models
clfs = [knn_best, svc_best, lr_best]
compare_clfs_perf(clfs, X_train, y_train, X_test, y_test, cv, scoring='accuracy', n_jobs=-1)
```

```
100%|██████████| 3/3 [01:26<00:00, 28.99s/it]
```

	clf name	clf params	clf validation accuracy std (CrossVal)	clf train accuracy mean (CrossVal)	clf validation accuracy mean (CrossVal)	accuracy (testing dataset)	clf time
0	LogisticRegression	{'C': 4, 'class_weight': None, 'dual': False, ...	0.00821657	0.894819	0.878264	0.887479	0.350565
1	SVC	{'C': 1000, 'break_ties': False, 'cache_size':...	0.0167968	0.898996	0.875923	0.887145	4.43243
2	KNeighborsClassifier	{'algorithm': 'auto', 'leaf_size': 30, 'metric...	0.0218767	0.878912	0.841063	0.856093	0.191535

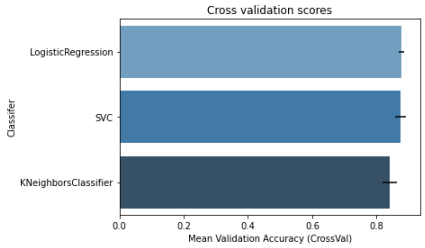

Figure 6.10: Overfitting checking of tuned models via cross validation

6.1.2.3. Final Ensemble Model

Since we got more than one model, we utilized the ensembling technique to integrate them and then got the final model. Its performance is shown in Figure 6.11. Compared to the models created based on the TF-IDF matrix (Table 6.2), this final ensemble model doesn't have significant improvement in terms of performance since they achieved similar accuracy, precision and recall.

```
# create final ensemble model

from modeling_and_evaluation import ensemble_best_clfs_and_predict

clfs_best = [('knn', knn_best), ('svc', svc_best), ('lr', lr_best)]
y_pred = ensemble_best_clfs_and_predict(clfs_best, X_train, y_train, X_test, y_test)
```

```
Detailed classification report:

              precision    recall  f1-score   support

           0       0.87      0.84      0.86      1175
           1       0.90      0.92      0.91      1820

    accuracy                           0.89      2995
   macro avg       0.89      0.88      0.88      2995
weighted avg       0.89      0.89      0.89      2995
```

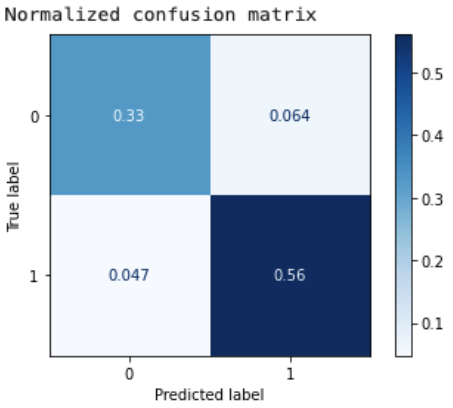

Figure 6.11: Performance of final ensemble model

Furthermore, a testing process with another new dataset was also conducted in order to validate the generalization of our final ensemble model. We first scrapped another 10k reviews from Google Play which were written during Feb. 1, 2020 and June 21, 2020. Then 2,500 reviews were randomly sampled from this new dataset and acted as a new testing set. With this new testing set our final model achieved 86% accuracy, 88% recall and 85% precision (Figure 6.12). Its performance decreased slightly compared to Figure 6.11 but still can be treated as stable. Hence, this final ensemble model is acceptable.

```
# validate the generalization of the final ensemble model via new testing set

X_test_final = pd.read_csv('../data/X_test_final.csv').to_numpy()
y_test_final = pd.read_csv('../data/y_test_final.csv')['0'].apply(lambda x: 1 if x==0 else 0).to_numpy()
y_pred = ensemble_best_clfs_and_predict(clfs_best, X_train, y_train, X_test_final, y_test_final)
```

```
Detailed classification report:

              precision    recall  f1-score   support

           0       0.86      0.83      0.85      1049
           1       0.85      0.88      0.87      1164

    accuracy                           0.86      2213
   macro avg       0.86      0.85      0.86      2213
weighted avg       0.86      0.86      0.86      2213
```

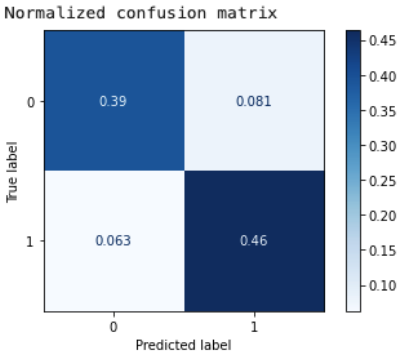

Figure 6.12: Validating the generalization of the final ensemble model with new testing dataset.

6.2 Aspect-based Sentiment Analysis

Lastly, we want to perform aspect extraction. In our project, we have applied topic modeling, which is an aspect extraction method for analyzing a large volume of non-annotated text. It is a text mining approach that uses statistical machine learning techniques to discover hidden patterns in order to create topics based on word groupings in text. It involves extracting the most representative topics occurring in a collection of documents and grouping the documents under a topic (Okafor, 2020). In our analysis we applied two topic modelling techniques: Latent Dirichlet Allocation (LDA) and Non-negative Matrix Factorization (NMF).

The input to topic modeling is bags of words, and the output is a classification, clustering, or feature extraction. In this project, we used an NMF model to display the topics and the weight for each word assigned to each topic. We used LDA for clustering and to visualize the clustered data.

Before we apply the topic modeling, we need to handle the text features using the feature function shown in the figure below. The feature function removes nouns such as “love” and “walmart” which are manually added to the remove_word list. For each sentence, the words are tokenized, stop words are removed from the wordList, and the NLP part of speech (POS) tagger is applied to the wordList. Then nouns are filtered

out from the word-tag pair and the list of results is appended to the new feature tagged_review (Figure 6.13).

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.datasets import fetch_20newsgroups
from sklearn.decomposition import NMF, LatentDirichletAllocation
import numpy as np
stop_words = set(stopwords.words('english'))
nltk.download('punkt')

def feature(df, feature):

    feature = feature.astype(str) # Dummy text

    # sent_tokenize is one of instances of
    # PunktSentenceTokenizer from the nltk.tokenize.punkt module
    res = []
    remove_word = ['love','walmart']
    for i in feature:
        unique = ' '.join(set(i.split(' ')))
        wordsList = nltk.word_tokenize(unique) #     Word tokenizers is used to find the words and punctuation in a string
        wordsList = [w for w in wordsList if not w in stop_words and not w in remove_word]  # removing stop words from wordList

        #  Using a Tagger. Which is part-of-speech tagger or POS-tagger.
        tagged = [nltk.pos_tag(wordsList)]
        for row in tagged:
            tmp = []
            for pair in row:
                word, tag = pair
                if tag == 'NN':
                    tmp.append(word)
            res.append(' '.join(tmp))
        df['tagged_review'] = pd.DataFrame(res)
    return df.head()
```

Figure 6.13: showing codes for handing features

Next, topic modeling is conducted and the display topics are derived. The display_topics function shows words and weights for each word in sentences and records. In topic Modeling, we apply both NMF and LDA models to the document-term matrix, and use the models to assign topics to papers in the corpus. Then we create features as NMF topic and LDA topic model, and we simply display the topic and the weighted scores for each word in sentence for selected models in Figure 6.14 and Figure 6.15.

```
def display_topics(model, feature_names, no_top_words):
    topic_dict = {}
    for topic_idx, topic in enumerate(model.components_):
        topic_dict["Topic %d words" % (topic_idx)]= ['{}'.format(feature_names[i]) for i in topic.argsort()[:-no_top_words - 1:-1]]
        topic_dict["Topic %d weights" % (topic_idx)]= ['{:.1f}'.format(topic[i]) for i in topic.argsort()[:-no_top_words - 1:-1]]
    return pd.DataFrame(topic_dict)
```

Figure 6.14: showing the helper function for displaying visualization about topic modeling

```
def topicModel(df, no_topics, no_top_words, model):
    documents = df['tagged_review']
    no_terms = 10000 #Set variable number of terms

    # NMF uses the tf-idf count vectorizer
    # Initialise the count vectorizer with the English stop words
    vectorizer = TfidfVectorizer(max_df=0.5, min_df=2, max_features=no_terms, stop_words='english')
    document_matrix = vectorizer.fit_transform(documents)     # Fit and transform the text
    feature_names = vectorizer.get_feature_names() #get features

        # Apply NMF topic model to document-term matrix
    nmf_model = NMF(n_components=no_topics, random_state=42, alpha=.1, l1_ratio=.5, init='nndsvd').fit(document_matrix)

        # Run LDA
    lda_model = LatentDirichletAllocation(n_components=no_topics, max_iter=5, learning_method='online', learning_offset=50.,random_state=0).fit(document_matrix)

        # Use NMF model to assign topic to papers in corpus
    nmf_topic_values = nmf_model.transform(document_matrix)
    lda_topic_values = lda_model.transform(document_matrix)
    df['NMF Topic'] = nmf_topic_values.argmax(axis=1)
    df['LDA Topic'] = lda_topic_values.argmax(axis=1)

    if model == 'NMF':
        return display_topics(nmf_model, feature_names, no_top_words)
    if model == 'LDA':
        return display_topics(lda_model, feature_names, no_top_words)
```

Figure 6.15: showing the step-by-step codes for topic modeling

We pick 5 topics and use 100 words for NMF and LDA. For each topic shown in Figure 6.16, every column of topic words is a group, and the weights are in descending order so that we can know which word makes the most contribution in each topic. Based on topic number and words` weight, we calculate NMF and LDA topic number for each review sentence (Figure 6.17).

```
feature(data, data['clean_sents'])
data = data.dropna()
topicModel(data, 5, 100, 'NMF')
```

	Topic 0 words	Topic 0 weights	Topic 1 words	Topic 1 weights	Topic 2 words	Topic 2 weights	Topic 3 words	Topic 3 weights	Topic 4 words	Topic 4 weights
0	work	8.2	order	7.2	use	7.6	time	7.4	item	6.2
1	check	0.1	grocery	2.0	pickup	0.2	fix	0.8	store	4.1
2	fine	0.1	place	1.2	grocery	0.2	waste	0.6	stock	1.6
3	need	0.1	pickup	0.9	pay	0.2	try	0.5	cart	0.8
4	update	0.1	pick	0.8	service	0.1	freeze	0.4	shop	0.7
5	grocery	0.1	online	0.7	week	0.1	crash	0.4	add	0.7
6	month	0.1	try	0.7	phone	0.1	delivery	0.4	price	0.6
7	stop	0.1	delivery	0.6	month	0.1	problem	0.2	check	0.6
8	reinstall	0.1	day	0.5	card	0.1	half	0.2	list	0.6
9	phone	0.1	check	0.5	year	0.1	check	0.2	need	0.5
10	scanner	0.1	problem	0.4	want	0.1	issue	0.2	try	0.5
11	version	0.0	week	0.3	update	0.1	load	0.1	search	0.4
12	week	0.0	cancel	0.3	shopping	0.1	pick	0.1	thing	0.4
13	thing	0.0	thing	0.3	location	0.1	slot	0.1	pickup	0.4
14	pay	0.0	hour	0.2	store	0.1	money	0.1	tell	0.3
15	skip	0.0	pay	0.2	option	0.1	phone	0.1	location	0.3
16	way	0.0	add	0.2	try	0.1	hour	0.1	delete	0.2
17	day	0.0	need	0.2	website	0.1	cart	0.1	look	0.2
18	pickup	0.0	let	0.2	shop	0.1	thing	0.1	let	0.2

Figure 6.16: showing the output of words and weight pairs in each topic

```
data.head()
```

	reviewId	score	review_sent	clean_sents	tagged_review	NMF Topic	LDA Topic
0	gp:AOqpTOHuC9a634QjathsyDrU3ip9At5azd7X8o6RDzf...	5	Love Walmart and the Walmart app too.	love walmart walmart		0	0
1	gp:AOqpTOHuC9a634QjathsyDrU3ip9At5azd7X8o6RDzf...	5	Lots of good stuff to buy here too.	lot good stuff buy	lot	2	0
2	gp:AOqpTOHuC9a634QjathsyDrU3ip9At5azd7X8o6RDzf...	5	Updates were nice also thanks so mooch..	update nice also thank mooch	thank update mooch	0	3
3	gp:AOqpTOF5XzVmyK52nMUPJO6R2RBU3SM241f5rO9PNTT...	5	My new way of shopping	new way shopping	way shopping	2	2
4	gp:AOqpTOEhCsdm2GlTL0qVcZdjVvPrziwp1isu_V2JmjM...	1	I hated the approach to accessing each departm...	hate approach access department	hate approach department access	0	0

Figure 6.17: showing the output for both NMF topic and LDA topic results

The sentence classification for NMF does not provide as much insight as LDA, so we need to expand upon the methods used. LDA takes a collection of documents and displays K-topics or clusters of words and for each topic we have a bar chart showing the related frequency of words as well as the distributions of topics (Figure 6.18).

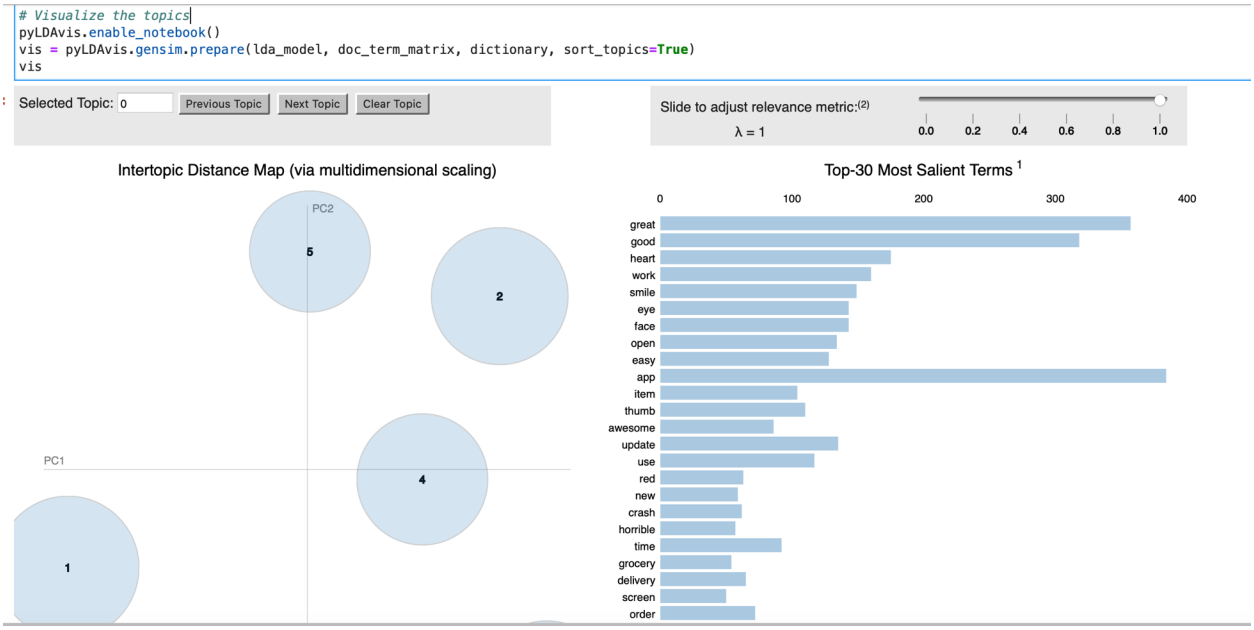

Figure 6.18: A visualization showing the LDA topic modelling results.

7. Future Work

In the future, there are three main areas that need to be improved based on the current results: sentiment analysis, deployment and text cleaning.

7.1 Sentiment Analysis

7.1.1 Aspect-based

Given that the result of the aspect extraction part is not sufficient, we need to try other extraction methods to get more accurate aspects. There are two main approaches that we can employ. The first is unsupervised methods as the raw reviews aren't labeled by aspects. These methods include rule-based models (Tripathi, 2019) and attention based models (Liang, 2020; Roever, 2020). We can also predefine some aspects and calculate the similarity of representative words in each review and these predefined aspects to derive each review's aspects based on unsupervised algorithms (Intellica.AI, 2020). Another option is to manually label a small section of the raw reviews with predefined aspects and then create a classifier to automatically label the remaining section (Min, 2019; Sam & Muktawar, 2019).

After extracting aspects, we will conduct aspect-based sentiment analysis using unsupervised methods as we won't have corresponding sentiment labels in each aspect that each raw review contains (Intellica.AI, 2020).

7.1.2 Sentiment Change Over Time

After acquiring the sentiment of reviews at different time points, we will analyze how customers' feelings about the product changes over time, consisting of both document-based sentiment and aspect-based sentiment. Through this we would gain insight into whether the customers' voices are heard effectively and if the product is increasingly improved.

7.1.3 Document-based

We have achieved an acceptable model to detect negative reviews timely through document-based sentiment analysis. However, there are also more advanced approaches utilizing deep learning algorithms to deal with this problem. We will try these methods and compare their performance with that of our current created model through ML algorithms. Then we could determine the final model according to their performance and time efficiency.

7.2 Deployment

So far, we have deployed but not created our classification model. To deploy it we need to complete the following three steps. First, we will build a web-based user interface to receive inputs and display outputs. A connection between this interface and our core model system will also be created so it can ingest incoming real-time data automatically. Then, we will establish an effective evaluation system to monitor the performance of this analysis system, which can respond promptly to significant (data or metrics) changes. Given that our goal is to deal with real time data, it is time sensitive so we also need to consider how to deploy it in a distributed environment.

7.3 Text Cleaning

Some improvements can be made to our text cleaning methods. For example, there are some spelling mistakes and common abbreviations (e.g. US) in the raw reviews, which we need to deal with to eliminate noise in the data. Moreover, the text cleaning pipeline takes an hour and half to clean the text. We need to utilize parallel programming and exploit some existing tools, such as the multiprocessing module, to take full advantage of all computer resources so as to improve the computational efficiency.

8. Conclusion

In summary, we tried two methods of conducting document-based sentiment analysis, TF-IDF and word embedding, and used several evaluation metrics to compare the outcome. The final ensemble model trained on the features extracted using word embedding is just slightly higher than the best accuracy model trained on features extracted using TF-IDF. Since the word embedding method did not result in a significant change in terms of accuracy, precision and recall, we can conclude that these two methods have similar performance for our dataset.

In our aspect extraction work, we found the sentence classification using NMF does not provide as much insight as LDA, so we used LDA which takes a collection of documents and displays K-topics or clusters of words.

Regarding future work, we can explore other extraction methods to get more accurate aspects and expand our work to a web-based user interface to give real time negative review alert and track the sentiment change over time.

With this sentiment analysis, we can gain insight into customers' attitudes towards the application. In the future, a system could be developed for companies to detect negative reviews in real time and identify areas of their product or service that they should improve.

References

Intellica.AI. (2020, February 26). Aspect-based Sentiment Analysis-Everything You Wanted to Know! Retrieved December 12, 2020, from https://medium.com/@Intellica.AI/aspect-based-sentiment-analysis-everything-you-wanted-to-know-1be41572e238

LIANG, X. (2020, May 31). Evaluation of Unsupervised Attention Model for Aspect Term Extraction. Retrieved December 11, 2020, from https://towardsdatascience.com/evaluation-of-unsupervised-attention-model-for-aspect-term-extraction-da887728fba8

Mikolov, T., Grave, E., Bojanowski, P., Puhrsch, C., & Joulin, A. (2017). Advances in pre-training distributed word representations. arXiv preprint arXiv:1712.09405.

Min, P. (2019, February 13). Aspect-Based Opinion Mining (NLP with Python). Retrieved December 12, 2020, from https://medium.com/@pmin91/aspect-based-opinion-mining-nlp-with-python-a53eb4752800

Okafor, O. (2020, October 10). Automatic topic classification of research papers using the NLP topic model NMF. Medium. https://medium.com/@obianuju.c.okafor/automatic-topic-classification-of-research-papers-using-the-nlp-topic-model-nmf-d4365987ec82

PlanB. (n.d.). Google-play-scraper. Retrieved October 14, 2020, from https://pypi.org/project/google-play-scraper/

Roever, S. (2020, August 10). Aspects, the better topics? Applying unsupervised aspect extraction on Amazon cosmetics reviews. Retrieved December 12, 2020, from https://medium.com/@sanne.de.roever/aspects-the-better-topics-applying-unsupervised-aspect-extraction-on-amazon-cosmetics-reviews-9d523747f8e5

Sam, S., & Muktawar, N. (2019, August 02). A Guide to Learning with Limited Labeled Data. Retrieved December 12, 2020, from https://blog.cloudera.com/a-guide-to-learning-with-limited-labeled-data/

Scott, W. (2019, May 21). TF-IDF for Document Ranking from scratch in python on real world dataset. Retrieved December 12, 2020, from https://towardsdatascience.com/tf-idf-for-document-ranking-from-scratch-in-python-on-real-world-dataset-796d339a4089

Tripathi, N. (2019, January 04). Aspect-Based Sentiment Analysis in Product Reviews: Unsupervised Way. Retrieved December 11, 2020, from https://medium.com/@nitesh10126/aspect-based-sentiment-analysis-in-product-reviews-unsupervised-way-fb0b38ead501