Edge Hill University

The Department of Computer Science

CIS4515
Practical Data Analysis

Level 7

Coursework 2

Report

2023/2024

Student: Joseph Z. Moyo

Student No: 25792334

Abstract

Keywords:

Contents

1. Introduction

provide background information; the purpose and objectives of the sentiment analysis task. Do not assume that the reader has specialised knowledge of the area that you are describing and be sure to explain any technical terms that you use.

2. Literature review

Sentiment analysis, often referred to as opinion mining, is a field of study that analyses people's sentiments, attitudes, and emotions towards entities such as products, services, and individuals (Jemai, Hayouni and Baccar, 2021). It involves collecting and examining subjective information from text data, primarily to understand and categorise the opinions expressed. Jemai, Hayouni and Baccar (2021) discuss various machine learning techniques for sentiment analysis, emphasising the importance of data preprocessing and feature extraction for enhancing model accuracy. Jemai, Hayouni and Baccar (2021) employ text mining techniques to process variables using the Natural Language Toolkit (NLTK) tools and a supervised probabilistic machine learning algorithm to classify tweets into positive or negative sentiments.

NLTK is a comprehensive Python package designed for natural language processing (NLP) tasks. It includes a variety of tools for classification, tokenisation, stemming, tagging, parsing, and semantic reasoning (Jemai, Hayouni and Baccar, 2021). NLTK is widely used for linguistic data analysis, education, and research. The library is open-source, making it freely accessible for developers and researchers to build NLP applications.

Tokenisation is the process of breaking down text into smaller units, called tokens, which can be words, phrases, or symbols, to facilitate easier analysis and processing in natural language processing tasks (Basa and Basarslan, 2023). After the text has been disintegrated to smaller fragments it is pos-tagged, a process of marking up words in a text as corresponding to a particular part of speech, based on both its definition and context (Basa and Basarslan, 2023). Subsequently the stopwords are removed. This involves eliminating common words that add little value in understanding the sentiment of the text, such as "the" and "is" (Basa and Basarslan, 2023). Finally, the lemmatisation process of reducing words to their base or dictionary form, preserving their meaning and context in natural language processing is executed (Basa and Basarslan, 2023).

Vectorisation is the process of representing text or data as numerical vectors, enabling computational analysis and machine learning (Basa and Basarslan, 2023). The authors used a vectorisation method called the Term Frequency-Inverse Document Frequency (TF-IDF), which is a frequency-based text representation technique for extracting the importance of words within a corpus of documents. TF-IDF helps in determining the relevance of a word to a document in a collection, which is useful for filtering out common words that are less informative. It reduces the feature space by considering only those terms that are most descriptive of the content, thus improving the efficiency of machine learning algorithms. The method is straightforward to implement and understand, making it accessible for various applications (Basa and Basarslan, 2023). However, TF-IDF does not account for the context or order of words, which can lead to a loss of semantic meaning (Bose and Roy, 2023). It may overemphasise rare terms, which are not always significant, potentially skewing the results. The method provides a static representation of text and does not adapt to new data or changes in language use over time (Bose and Roy, 2023). Overall, while TF-IDF is a powerful tool for initial text analysis, its limitations suggest that it may be complemented with more advanced techniques, such as word embeddings or deep learning models, for a more nuanced understanding of text data.

Biruntha, Arul and Ashwin (2022) used the CountVectorizer for vectorisation of their text data. This technique converts text data into a matrix of token counts, where each row represents a document (or text sample) and each column corresponds to a unique word. CountVectorizer is straightforward to implement and understand. It creates a bag-of-words representation, which is easy to interpret. It efficiently handles large text corpora by creating a sparse matrix with word frequencies. It serves as a baseline for more advanced techniques. It captures the frequency of words, which can be useful for initial exploration (Biruntha, Arul and Ashwin, 2022). According to Pavitha et al (2022), the downside is that the CountVectorizer treats all words equally, regardless of their context or importance. Rare words and common stopwords receive the same weight. This suggests that it lacks semantic understanding; it does not capture word meanings or relationships. The resulting feature matrix can be high-dimensional, leading to computational challenges and potential overfitting (Pavitha et al, 2022). It includes noisy features (e.g., misspellings, typos) that might not contribute meaningfully. While CountVectorizer is a useful starting point, researchers often combine it with other techniques (e.g., TF-IDF, word embeddings) to address its limitations and enhance text representation for machine learning tasks.

Both CountVectorizer and TF-IDF techniques can be combined to create a feature representation that captures both word counts and the importance of words (Ganesan, 2019). The CountVectorizer can be applied first to obtain the word count vectors. Then followed up with a TfidfTransformer to compute the TF-IDF scores based on the word counts. The resulting feature vectors will include both raw counts and weighted scores. This combined approach can be beneficial in various NLP tasks, such as text classification, clustering, or information retrieval (Ganesan, 2019).

Jemai, Hayouni and Baccar (2021) use various classifiers to for classifying text data post vectorisation. One of these is the Multinomial Naïve Bayes (MNB). It works well with discrete features and is suitable for building feature vectors representing the frequency of occurrence. Jemai, Hayouni and Baccar (2021) reports high accuracy levels using MNB, indicating its effectiveness in classifying sentiments. Its limitations identified include the MNB's assumption that all features are independent, which may not hold true in real-world data, potentially affecting the classifier's performance. The algorithm performs better with larger vocabulary sizes, which implies a need for substantial training data. Jemai, Hayouni and Baccar (2021) mentions the model's failure to detect complex sentiments like sarcasm, indicating a limitation in handling nuanced expressions. The authors demonstrate that while MNB can be highly effective for sentiment analysis, it also has inherent limitations that must be considered during implementation.

Jemai, Hayouni and Baccar (2021) report that logistic regression achieved an accuracy of about 86.23% when using the bigram model, which is higher compared to other supervised machine learning algorithms for sentiment analysis on Twitter. Logistic regression provides probabilities for class memberships, offering a measure of certainty about the classification (Jemai, Hayouni and Baccar, 2021). It is easy to interpret the model coefficients, which can provide insights into the importance of unique features. Limitations identified include the logistic regression assumption of linear decision boundaries, which may not capture complex relationships in the data as effectively as non-linear models. The performance heavily relies on the choice of features; the bigram model showed better results, indicating the importance of feature engineering (Jemai, Hayouni and Baccar, 2021).

Basa and Basarslan (2023) employ various classifiers including the Random Forest (RF). RF combines multiple decision trees to improve accuracy and prevent overfitting. It handles large datasets with higher dimensionality well. RF showed a high accuracy of 86% in the study, indicating its effectiveness in sentiment classification. Its limitations include its complexity and computational intensiveness due to the use of many decision trees. It is harder to interpret the results of RF compared to simpler models like Decision Trees. If the training data is biased, RF can overfit to these biases, affecting the generalizability of the model. Basa and Basarslan (2023) suggests that while RF is a powerful tool for sentiment analysis, careful consideration must be given to its complexity and the quality of the training data.

Voting classifiers are ensemble learning models that combine predictions from multiple machine learning algorithms to make a final decision. They operate by aggregating the outputs of individual classifiers and selecting the class with the majority vote or the highest probability (Bandi et al, 2023). These classifiers can use 'hard' voting, which relies on the predicted class labels, or 'soft' voting, which considers the probability estimates for each class. They are versatile and can integrate various base learners like Decision Trees, KNN, and Random Forest. The primary advantage of voting classifiers is their ability to improve prediction accuracy by leveraging the strengths of multiple models (Bandi et al, 2023). They are robust against overfitting and often perform better than any single classifier, especially when the base learners are diverse. However, voting classifiers can be computationally expensive due to the need to train multiple models. They also require careful tuning to balance the contribution of each base learner and may not perform well if the individual classifiers are too correlated or if there is a significant disparity in their performance (Bandi et al, 2023).

SMOTE, or Synthetic Minority Oversampling Technique, is a widely used approach to address the imbalance in datasets where one class significantly outnumbers another. It generates synthetic examples by interpolating between minority class examples and their nearest neighbours (Chen et al, 2022). Utilizes k-nearest neighbours to create new, synthetic minority class examples. SMOTE can be integrated with various classifiers and is adaptable to different datasets. Its main advantage is its aim to balance class distribution, reducing classifier bias towards the majority class. By creating more examples, it helps classifiers better generalise to unseen data. However, it can introduce noisy examples if the parameter k is not optimally set (Chen et al, 2022). It may exacerbate the overlapping of classes, leading to ambiguous decision boundaries and there is a risk of creating duplicate examples which do not contribute to classifier performance (Chen et al, 2022).

Wu et al (2022) introduces HG-BERT, an optimized BERT model tailored for multimodal sentiment analysis. BERT, short for Bidirectional Encoder Representations from Transformers, stands as a state-of-the-art neural architecture introduced by Google. Its bidirectional nature enables it to capture contextual information from both preceding and subsequent tokens, thereby enhancing its ability to discern intricate nuances in language (Wu et al, 2022). From the study it is shown that the proposed HG-BERT model augments BERT's capabilities through several key innovations. First, it employs a hierarchical multi-head self-attention mechanism, allowing it to focus on relevant segments of input data. Second, a gate channel module filters out noise, enhancing the model's robustness. Lastly, a tensor fusion model facilitates effective feature fusion across modalities. Notably, HG-BERT exhibits a commendable accuracy improvement of 5-6% over conventional models when evaluated on the CMU-MOSI dataset. However, like any technology, it bears limitations. Its computational demands, stemming from the large-scale pretraining, may hinder real-time deployment. Additionally, the model's interpretability remains a challenge, given its complex architecture. Nevertheless, HG-BERT represents a significant stride in multimodal sentiment analysis, bridging the gap between language and emotion.

He and Hu (2022) propose the Adapted Multimodal BERT (AMB), addressing the challenge of multimodal sentiment analysis. It leverages pretrained language models, specifically BERT, which captures semantic and context-aware features from textual data. AMB combines adapter modules and intermediate fusion layers to enhance its performance on multimodal tasks. These components fine-tune the pretrained language model for the specific task, allowing for efficient adaptation without altering the entire model. These layers perform layer-wise fusion of audio-visual information with textual BERT representations (He and Hu, 2022). This multimodal fusion approach enables better integration of nonverbal cues (acoustic and visual) with language features. It is evident from the study that by keeping the pre-trained language model parameters frozen during adaptation, AMB achieves fast and efficient training. AMB demonstrates robustness to input noise, making it suitable for real-world scenarios. Experimental results on sentiment analysis using the CMU-MOSEI dataset reveal that AMB outperforms the current state-of-the-art, achieving a 3.4% relative reduction in error and a 2.1% relative improvement in 7-class classification accuracy. However, while AMB improves performance, it does so at the cost of increased model parameters due to the combination of adapters and fusion layers. The effectiveness of AMB may vary across different multimodal tasks, necessitating further investigation and adaptation for specific domains.

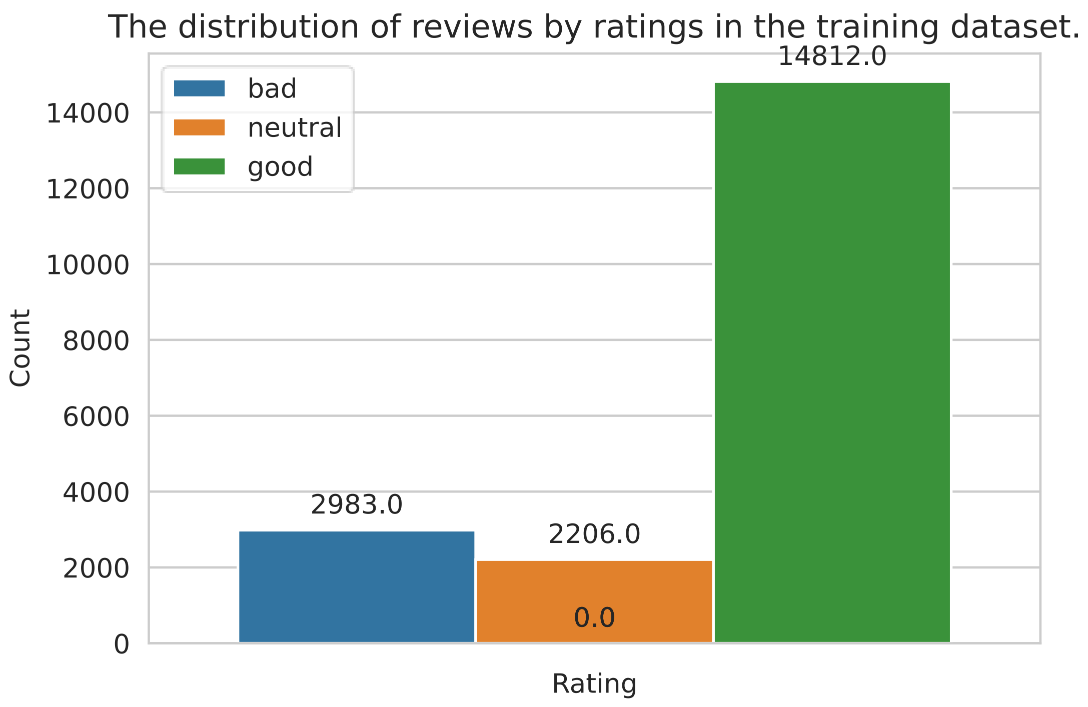
3. Methodology

The sentiment analysis task for identifying the best android application from the Amazon reviews test was approached using a data mining workflow that incorporated feature engineering by tokenisation of raw text, followed by pos-tagging, filtering of English stopwords, lemmatisation of tokens to their root and subsequently vectorising the string data to numerical data using CountVectorisor in conjunction with a tfidfTransformer, and ultimately training various ML models to observe their predictive performances. A GridSearch algorithm in python was utilised to experimentally obtain the optimum parameters of individual ML classifiers considered for assembling a voting classifier. Another voting classifier was constructed in parallel, consisting of numerous versions of the same base classifiers but with varying parameters to increase robustness and decrease susceptibility to overfitting. The two voting classifiers were compared and tested on the testing dataset to establish their performance. Ultimately the best performing ensemble classifier was used to evaluate which application producing company the investment funds would be best allocated to.

3.1 EDA

The training and test datasets each contain three feature columns namely, ‘rating’, ‘app’ and ‘review’. The training dataset has a total of 20 001 reviews and their distribution by rating is presented in Figure 1.

Figure 1: The distribution of reviews by ratings in the training dataset.



Most of the reviews in the training dataset are overwhelmingly rated good. The test dataset has a total of 19 999 reviews.

3.2 Libraries, Modules, Algorithms

The task in its entirety was completed using the python Jupyter Notebooks running on Google cloud computing via Google Colab. The libraries, modules and functions used were imported as shown in the screenshot below. These range from visualisation libraries like seaborn and matplotlib, to machine learning algorithms like the Logistic Regression and Random Forest from the Scikit Learn Library.

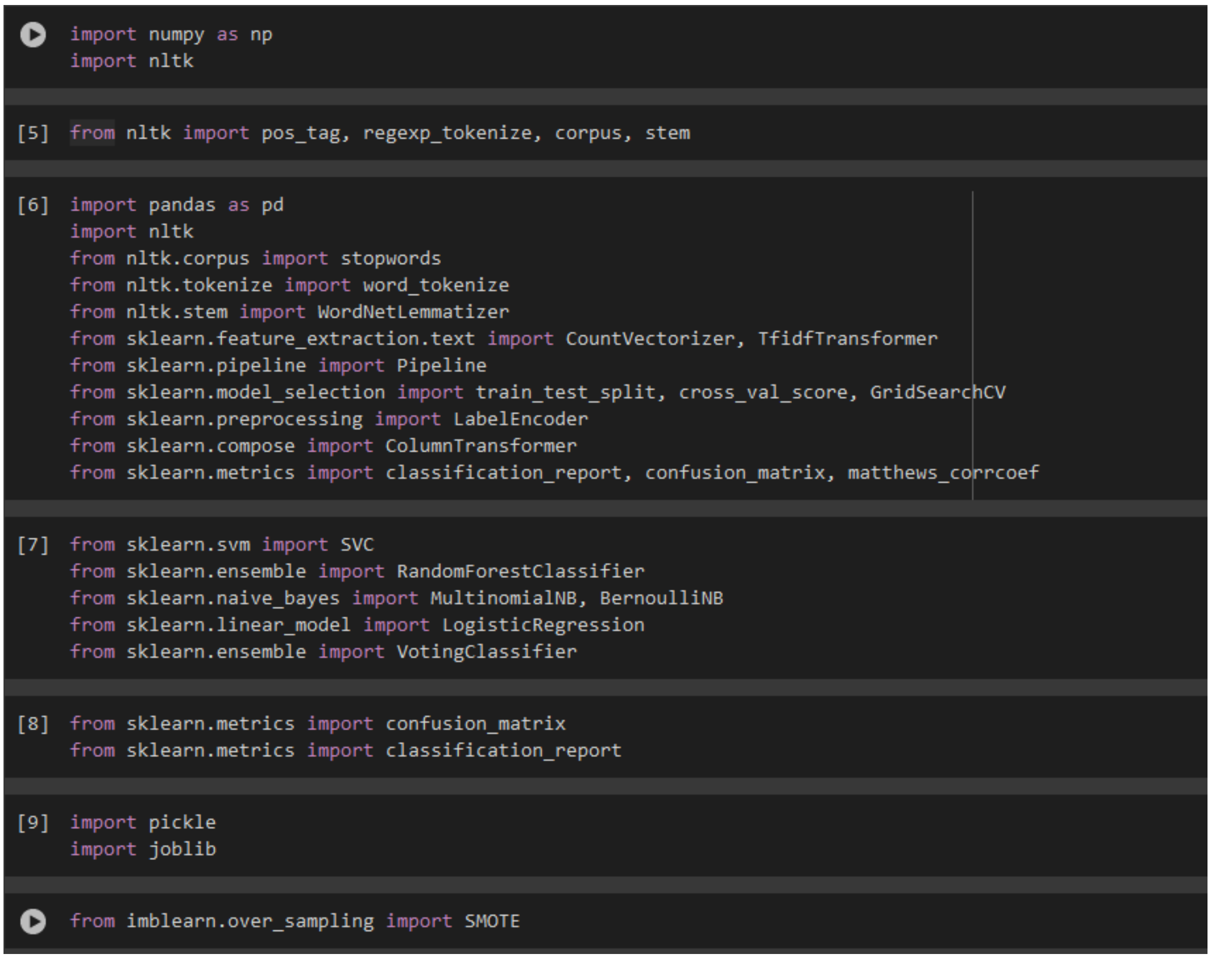

```
import numpy as np
import nltk

[5] from nltk import pos_tag, regexp_tokenize, corpus, stem

[6] import pandas as pd
    import nltk
    from nltk.corpus import stopwords
    from nltk.tokenize import word_tokenize
    from nltk.stem import WordNetLemmatizer
    from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
    from sklearn.pipeline import Pipeline
    from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
    from sklearn.preprocessing import LabelEncoder
    from sklearn.compose import ColumnTransformer
    from sklearn.metrics import classification_report, confusion_matrix, matthews_corrcoef

[7] from sklearn.svm import SVC
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.naive_bayes import MultinomialNB, BernoulliNB
    from sklearn.linear_model import LogisticRegression
    from sklearn.ensemble import VotingClassifier

[8] from sklearn.metrics import confusion_matrix
    from sklearn.metrics import classification_report

[9] import pickle
    import joblib

from imblearn.over_sampling import SMOTE
```

Table 1 shows the summary of the ML algorithms that were used in this task, highlighting their attributes, advantages and limitations. Prior to the ML task implementation, the android application reviews raw text data was feature engineered to tokenise, pos-tagged and processed into vectors that are machine legible. In the following Experiment section, it is documented in detail how the process was approached.

Table 1: ML algorithms used to analyse the android application reviews dataset (Bandi et al, 2023; Basa and Basarslan, 2023; Jemai, Hayouni and Baccar, 2021).

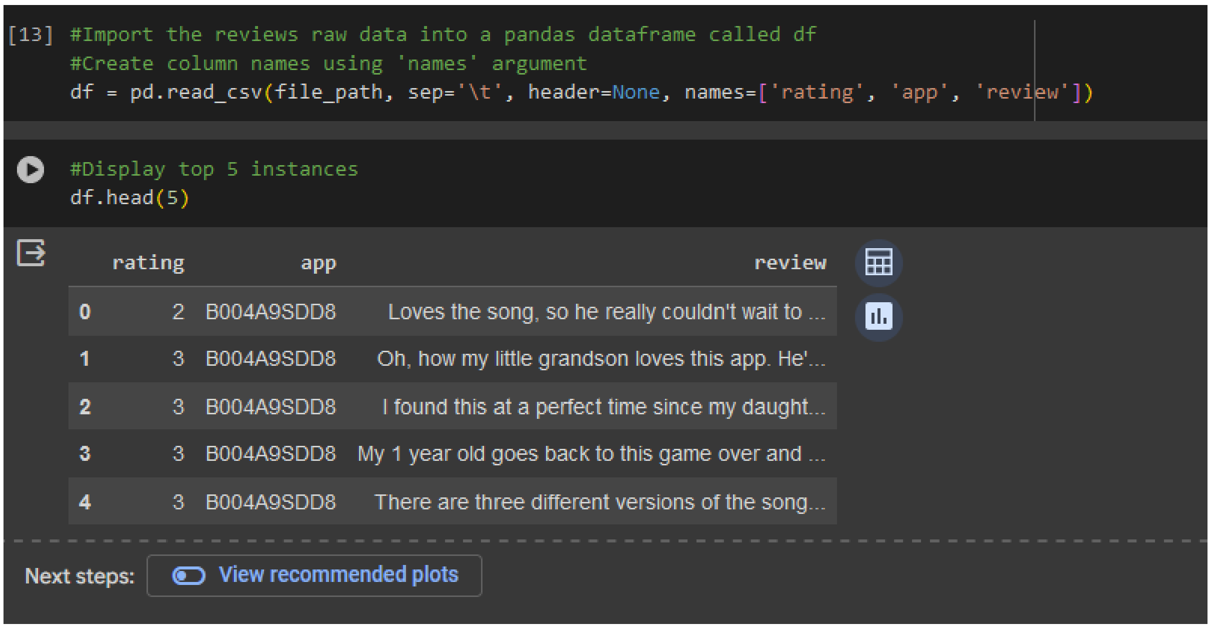
ML algorithm	Description	Attributes	Advantages	Limitations
Multinomial Naive Bayes (MNB)	MNB is a probabilistic classifier based on Bayes' theorem. It assumes that features are conditionally independent given the class label. Commonly used for text classification tasks	**Interpolation Method**: Utilizes k-nearest neighbours to create synthetic examples. **Flexibility**: Can be integrated with various classifiers and is adaptable to different datasets.	**Reduces Bias**: Balances class distribution, reducing classifier bias towards the majority class. **Enhances Generalization**: Creates more examples, helping classifiers generalize better.	**Potential Noise**: May introduce noisy examples if the parameter k is not optimized. **Overlapping Classes**: Can exacerbate class overlap, leading to ambiguous decision boundaries.

	(e.g., spam detection, sentiment analysis).			**Duplicate Examples**: Risk of creating nearly identical examples that don't improve performance.
Random Forest	Ensemble method that builds multiple decision trees and combines their predictions. Each tree is trained on a random subset of data (bootstrap samples) and features.	**Robust**: Handles non-linear relationships and reduces overfitting. **Parallelisable**: Trees can be built in parallel.	**Robustness**: Combines predictions from multiple trees, reducing individual tree biases. **Feature Importance**: Provides feature importance scores.	**Computationally Expensive**: Building multiple trees can be resource intensive. **Interpretability**: Harder to interpret than single decision trees.
Logistic Regression	Linear model used for binary classification (can be extended to multi-class). Learns a linear decision boundary by minimizing the logistic loss function.	**Interpretable**: Simple model with interpretable coefficients. **Linear Relationship**: Assumes linear relationship between features and log-odds.	**Simplicity**: Easy to understand and implement. **Interpretability**: Coefficients indicate feature importance.	**Linear Assumption**: Sensitive to deviations from linear relationships. **Outliers**: Prone to outliers affecting the decision boundary.
Voting Classifier	The Voting Classifier combines predictions from multiple base classifiers to make a final decision. It can be used for both binary and multi-class classification tasks. Two common voting methods: **hard voting** (majority vote) and **soft voting** (weighted average probabilities).	**Base Classifiers**: The Voting Classifier integrates several base classifiers (e.g., Multinomial Naive Bayes, Random Forest, Logistic Regression). **Voting Method**: Soft voting (weighted average probabilities) or hard voting (majority vote).	**Diverse Ensemble**: Combines predictions from different classifiers, reducing individual biases. **Robustness**: Less sensitive to overfitting compared to individual classifiers.	**Complexity**: The Voting Classifier introduces additional complexity due to combining multiple models. **Correlated Base Classifiers**: If base classifiers are highly correlated, the ensemble may not perform well.

4. Experiments

4.1 Feature Engineering

The raw reviews text data was imported into a pandas dataframe called 'df' for efficient querying and application of feature engineering functions. The dataframe initially consisted of three columns namely, 'rating', 'app' and 'review', with each row representing a single review as shown in the screenshot below.

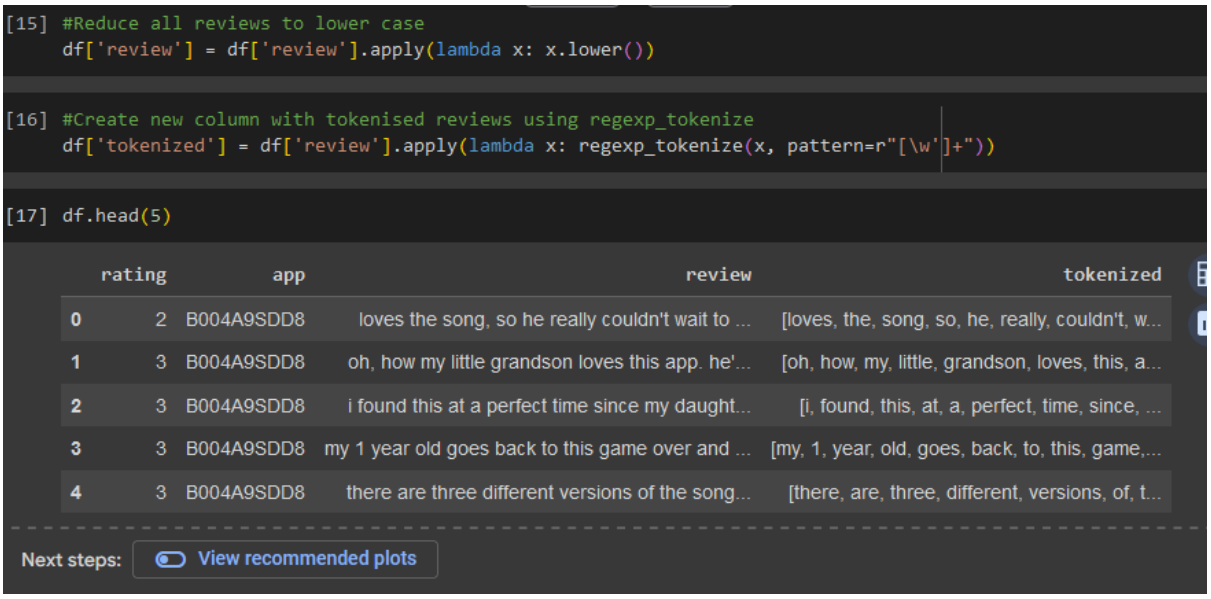


```
[13] #Import the reviews raw data into a pandas dataframe called df
     #Create column names using 'names' argument
     df = pd.read_csv(file_path, sep='\t', header=None, names=['rating', 'app', 'review'])
```

```
#Display top 5 instances
df.head(5)
```

```
   rating         app                                         review
0       2  B004A9SDD8    Loves the song, so he really couldn't wait to ...
1       3  B004A9SDD8   Oh, how my little grandson loves this app. He'...
2       3  B004A9SDD8     I found this at a perfect time since my daught...
3       3  B004A9SDD8  My 1 year old goes back to this game over and ...
4       3  B004A9SDD8   There are three different versions of the song...
```

The python lambda function was applied on the dataframe to reduce each review to lower case text to eliminate aliasing of words. Lambda was also used to apply tokenisation on the reviews using the 'regexp_tokenize()' function from the NLTK library. This tokenisation method does not split words with apostrophes when the argument, *pattern = r"[\w']+"*, is passed. The tokenised words were added onto a separate column in the dataframe. The screenshot below shows the application and result.



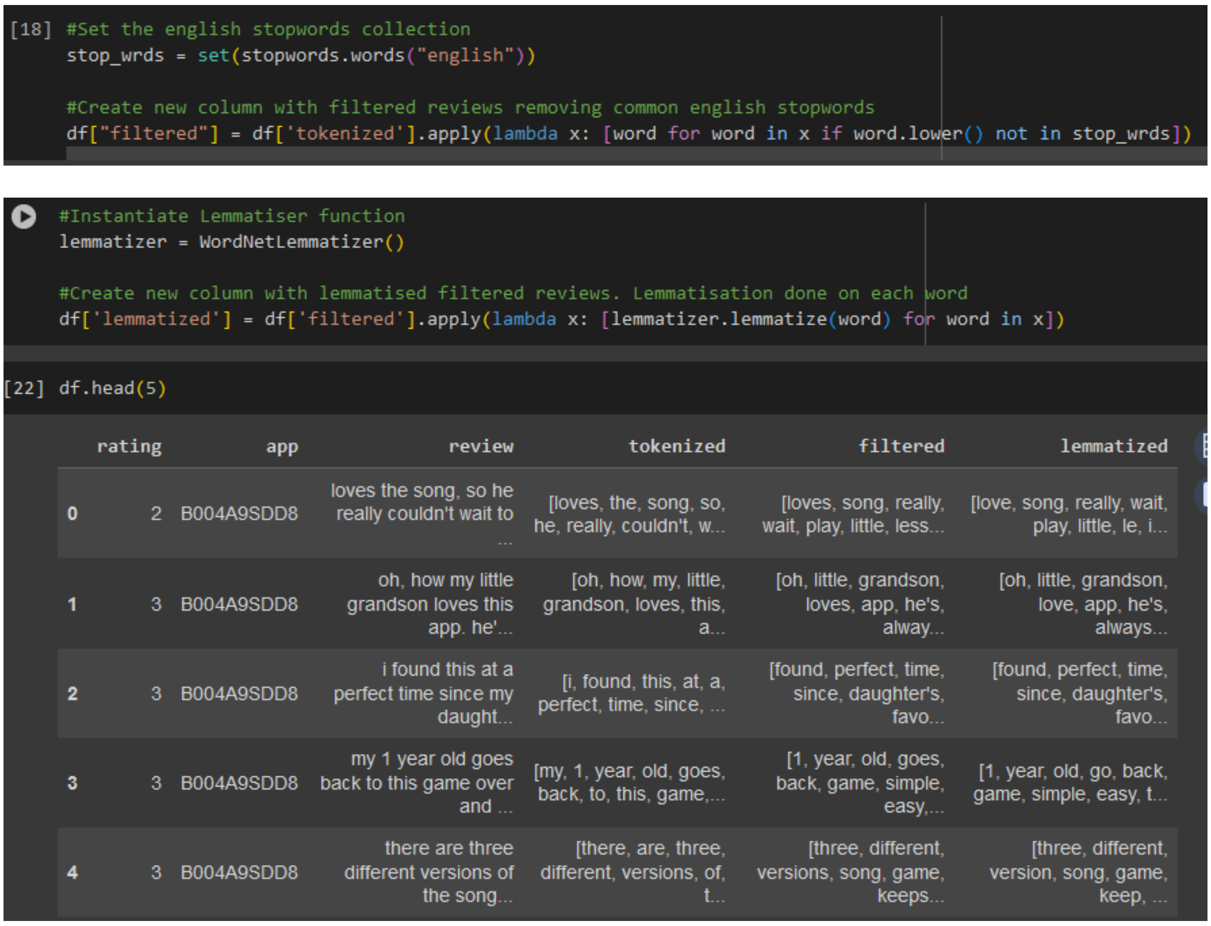
```
[15] #Reduce all reviews to lower case
     df['review'] = df['review'].apply(lambda x: x.lower())
```

```
[16] #Create new column with tokenised reviews using regexp_tokenize
     df['tokenized'] = df['review'].apply(lambda x: regexp_tokenize(x, pattern=r"[\w']+"))
```

```
[17] df.head(5)
```

```
   rating  app         review                                              tokenized
0  2       B004A9SDD8  loves the song, so he really couldn't wait to ...   [loves, the, song, so, he, really, couldn't, w...
1  3       B004A9SDD8  oh, how my little grandson loves this app. he'...   [oh, how, my, little, grandson, loves, this, a...
2  3       B004A9SDD8  i found this at a perfect time since my daught...   [i, found, this, at, a, perfect, time, since, ...
3  3       B004A9SDD8  my 1 year old goes back to this game over and ...   [my, 1, year, old, goes, back, to, this, game,...
4  3       B004A9SDD8  there are three different versions of the song...  [there, are, three, different, versions, of, t...
```

Similarly, two more columns were created containing the stopword-filtered tokens and lemmatised filtered tokens as shown below.



```
[18] #Set the english stopwords collection
     stop_wrds = set(stopwords.words("english"))

     #Create new column with filtered reviews removing common english stopwords
     df["filtered"] = df['tokenized'].apply(lambda x: [word for word in x if word.lower() not in stop_wrds])
```

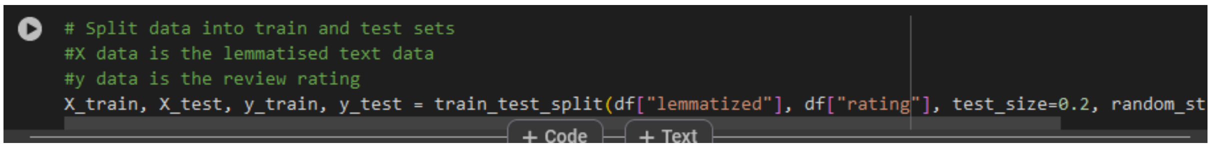
```
#Instantiate Lemmatiser function
lemmatizer = WordNetLemmatizer()

#Create new column with lemmatised filtered reviews. Lemmatisation done on each word
df['lemmatized'] = df['filtered'].apply(lambda x: [lemmatizer.lemmatize(word) for word in x])
```

```
[22] df.head(5)
```

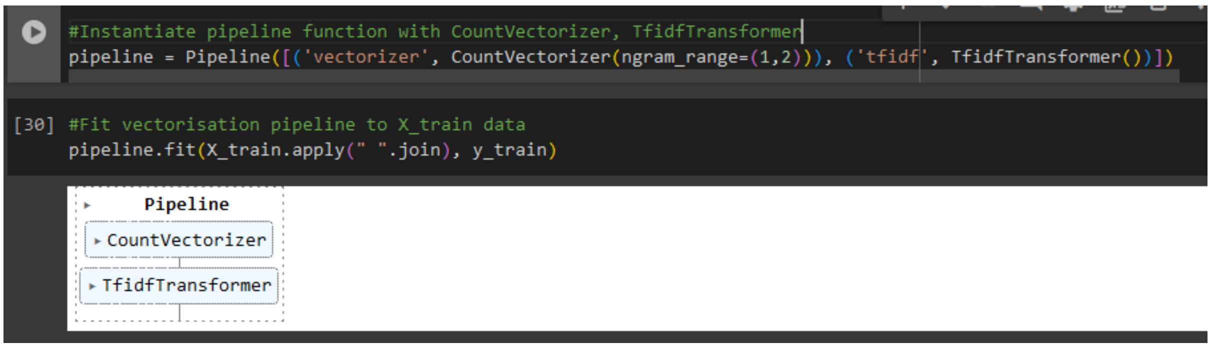
```
   rating  app         review                     tokenized                   filtered                    lemmatized
0  2       B004A9SDD8  loves the song, so he      [loves, the, song, so,      [loves, song, really,       [love, song, really, wait,
                       really couldn't wait to    he, really, couldn't, w...  wait, play, little, less...  play, little, le, i...
                       ...
1  3       B004A9SDD8  oh, how my little          [oh, how, my, little,       [oh, little, grandson,      [oh, little, grandson,
                       grandson loves this        grandson, loves, this,      loves, app, he's,           love, app, he's,
                       app. he'...                a...                        alway...                    always...
2  3       B004A9SDD8  i found this at a          [i, found, this, at, a,     [found, perfect, time,      [found, perfect, time,
                       perfect time since my      perfect, time, since, ...   since, daughter's,          since, daughter's,
                       daught...                                              favo...                     favo...
3  3       B004A9SDD8  my 1 year old goes         [my, 1, year, old, goes,    [1, year, old, goes,        [1, year, old, go, back,
                       back to this game over     back, to, this, game,...    back, game, simple,         game, simple, easy, t...
                       and ...                                                easy,...
4  3       B004A9SDD8  there are three            [there, are, three,         [three, different,          [three, different,
                       different versions of      different, versions, of,    versions, song, game,       version, song, game,
                       the song...                t...                        keeps...                    keep, ...
```

In the snippet above it can be observed that that words like “my”, “I”, “the”, and more have been removed in the ‘filtered’ column. The ‘lemmatised’ column on row index 3 has the word “goes” reduced to its root “go”.

The vectorisation of text or “string” data was carried out using the ‘CountVectorizer()’ function from SKlearn library in combination with the ‘TfidfTransformer()’ function. These two were implemented in a chain using the ‘pipeline()’ function from SKlearn as shown below. In the code presented, the training and testing portions of the dataset were partitioned using the ‘train_test_split()’ function from the SKlearn library, with 80 % of the data set aside for training and the remainder held back for testing.

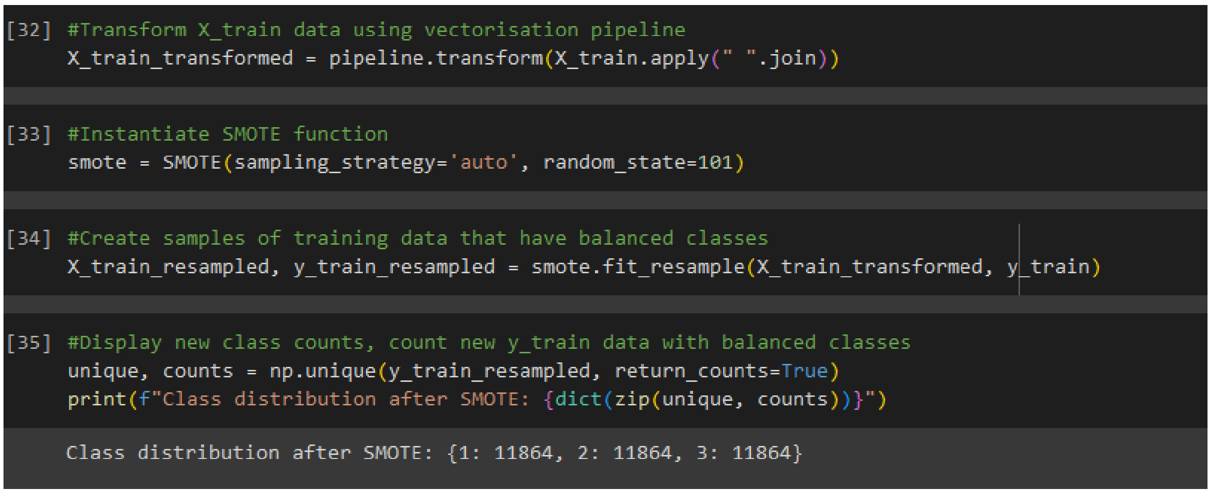


```
# Split data into train and test sets
#X data is the lemmatised text data
#y data is the review rating
X_train, X_test, y_train, y_test = train_test_split(df["lemmatized"], df["rating"], test_size=0.2, random_st
```



```
#Instantiate pipeline function with CountVectorizer, TfidfTransformer
pipeline = Pipeline([('vectorizer', CountVectorizer(ngram_range=(1,2))), ('tfidf', TfidfTransformer())])

[30] #Fit vectorisation pipeline to X_train data
     pipeline.fit(X_train.apply(" ".join), y_train)
```

The training dataset that was provided showed an overwhelming number of good reviews compared to neutral and bad. This skewed dataset would result in severe bias issues in the ML section of the task. As a mitigation action, the SMOTE technique was employed to balance the training dataset by creating synthetic bad and neutral reviews, ensuring an even distribution across the three classes. The technique was applied as shown in the screenshots below.



```
[32] #Transform X_train data using vectorisation pipeline
     X_train_transformed = pipeline.transform(X_train.apply(" ".join))

[33] #Instantiate SMOTE function
     smote = SMOTE(sampling_strategy='auto', random_state=101)

[34] #Create samples of training data that have balanced classes
     X_train_resampled, y_train_resampled = smote.fit_resample(X_train_transformed, y_train)

[35] #Display new class counts, count new y_train data with balanced classes
     unique, counts = np.unique(y_train_resampled, return_counts=True)
     print(f"Class distribution after SMOTE: {dict(zip(unique, counts))}")

     Class distribution after SMOTE: {1: 11864, 2: 11864, 3: 11864}
```

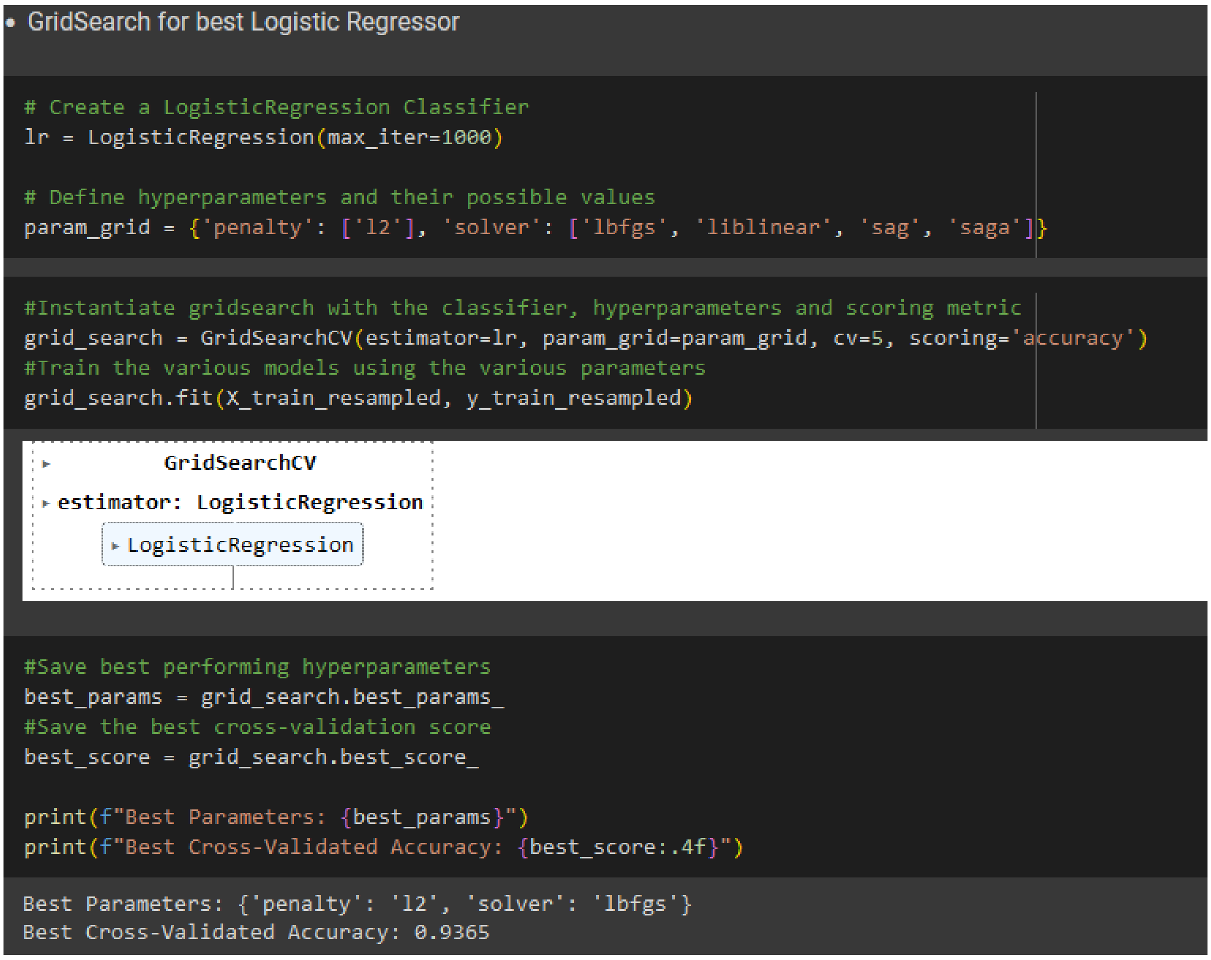
The bottom of the screenshot shows the new balanced class distribution for the training portion of the dataset. The balanced training portion of the dataset now had the labels negative (1), positive (3) and neutral (2), each having 11 864 instances.

4.2 Machine Learning

The Machine Learning tasks were partitioned into two exercises. The first pertaining to performing several GridSearch runs for obtaining the best parameters for the logistic regressor, random forest and Multinomial Naïve Bayes models. The second activity involved constructing two voting classifiers, with one consisting of the best parameter models from exercise one and the other comprising of 37 assorted variations of the three classifiers in the prior exercise.

4.2.1 Logistic regression

The LogisticRegression() model was instantiated and ran in its default state to get a first benchmark score from the balanced dataset, giving an accuracy score of 79 %. The GridSearch run was then conducted with the parameters as shown in the screenshot below. The best cross-validation accuracy score of 93.7 % was obtained from the parameter combination ‘penalty’ : ‘l2’ and ‘solver’ : ‘lbfgs’. These parameters were then noted for addition to the VC of best performing classifiers.


- GridSearch for best Logistic Regressor

```
# Create a LogisticRegression Classifier
lr = LogisticRegression(max_iter=1000)

# Define hyperparameters and their possible values
param_grid = {'penalty': ['l2'], 'solver': ['lbfgs', 'liblinear', 'sag', 'saga']}
```

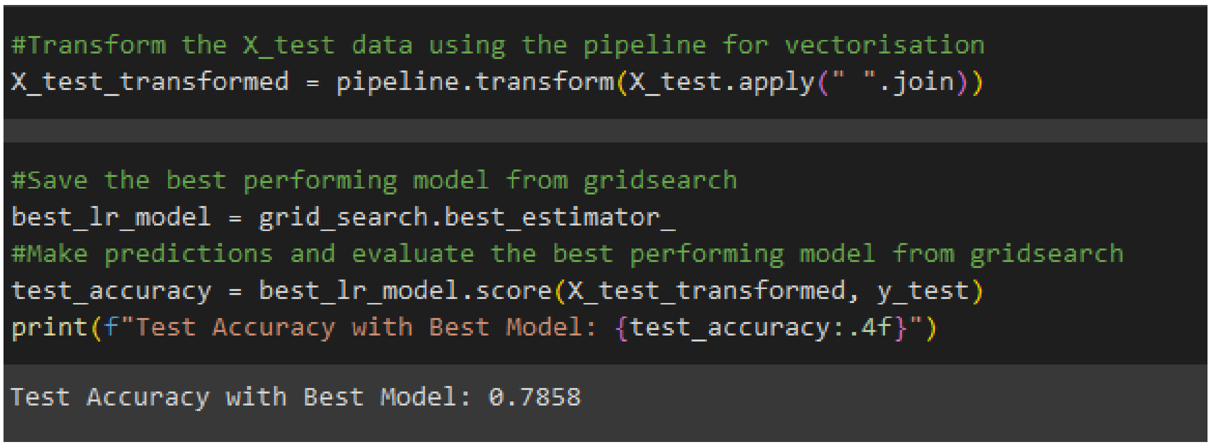
```
#Instantiate gridsearch with the classifier, hyperparameters and scoring metric
grid_search = GridSearchCV(estimator=lr, param_grid=param_grid, cv=5, scoring='accuracy')
#Train the various models using the various parameters
grid_search.fit(X_train_resampled, y_train_resampled)
```

```
▸          GridSearchCV
▸ estimator: LogisticRegression
      ▸ LogisticRegression
```

```
#Save best performing hyperparameters
best_params = grid_search.best_params_
#Save the best cross-validation score
best_score = grid_search.best_score_

print(f"Best Parameters: {best_params}")
print(f"Best Cross-Validated Accuracy: {best_score:.4f}")
```

```
Best Parameters: {'penalty': 'l2', 'solver': 'lbfgs'}
Best Cross-Validated Accuracy: 0.9365
```

The best performing parameters were evaluated on the training dataset only to give an accuracy score of 78.6 %, which was significantly lower than the cross-validation score previously obtained.

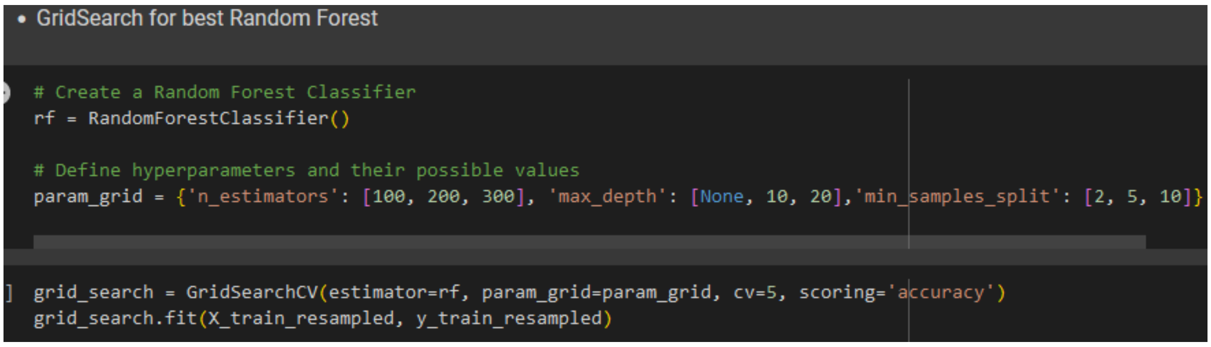

```
#Transform the X_test data using the pipeline for vectorisation
X_test_transformed = pipeline.transform(X_test.apply(" ".join))
```

```
#Save the best performing model from gridsearch
best_lr_model = grid_search.best_estimator_
#Make predictions and evaluate the best performing model from gridsearch
test_accuracy = best_lr_model.score(X_test_transformed, y_test)
print(f"Test Accuracy with Best Model: {test_accuracy:.4f}")
```

```
Test Accuracy with Best Model: 0.7858
```

4.2.2 Random Forest

A similar approach was done for the random forest runs and an accuracy score of 77 % was achieved using the default model that has 100 trees. The GridSearch was conducted with the params shown in the screenshot below. The best cross-validation accuracy observed was 96.4 %, with parameters: ‘n_estimators’: 300, ‘max_depth’ : ‘None’ and ‘min_samples_split’ : ‘2’.



• GridSearch for best Random Forest

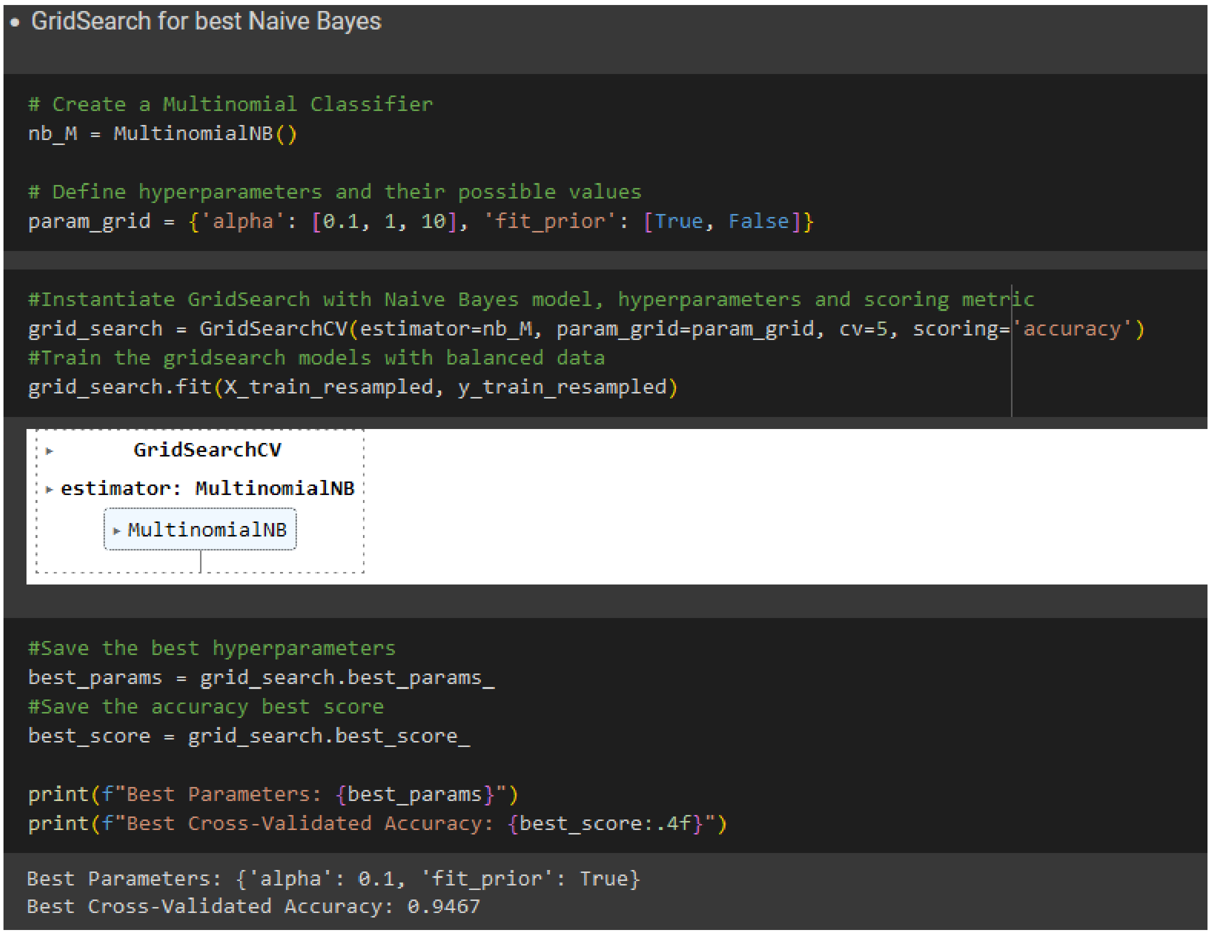
```
# Create a Random Forest Classifier
rf = RandomForestClassifier()

# Define hyperparameters and their possible values
param_grid = {'n_estimators': [100, 200, 300], 'max_depth': [None, 10, 20],'min_samples_split': [2, 5, 10]}
```

```
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train_resampled, y_train_resampled)
```

The model evaluation with the best parameter combination gave an accuracy score of 78.2 %.

4.2.3 Multinomial Naïve Bayes

The Multinomial Naïve Bayes model’s default setting gave an accuracy score of 74 %, which was the lowest of the three classifiers. The GridSearch parameters explored are presented below, and the best performing combination was 'alpha': 0.1, 'fit_prior': ‘True’.



• GridSearch for best Naive Bayes

```
# Create a Multinomial Classifier
nb_M = MultinomialNB()

# Define hyperparameters and their possible values
param_grid = {'alpha': [0.1, 1, 10], 'fit_prior': [True, False]}
```

```
#Instantiate GridSearch with Naive Bayes model, hyperparameters and scoring metric
grid_search = GridSearchCV(estimator=nb_M, param_grid=param_grid, cv=5, scoring='accuracy')
#Train the gridsearch models with balanced data
grid_search.fit(X_train_resampled, y_train_resampled)
```

```
▸ GridSearchCV
▸ estimator: MultinomialNB
    ▸ MultinomialNB
```

```
#Save the best hyperparameters
best_params = grid_search.best_params_
#Save the accuracy best score
best_score = grid_search.best_score_

print(f"Best Parameters: {best_params}")
print(f"Best Cross-Validated Accuracy: {best_score:.4f}")
```

```
Best Parameters: {'alpha': 0.1, 'fit_prior': True}
Best Cross-Validated Accuracy: 0.9467
```

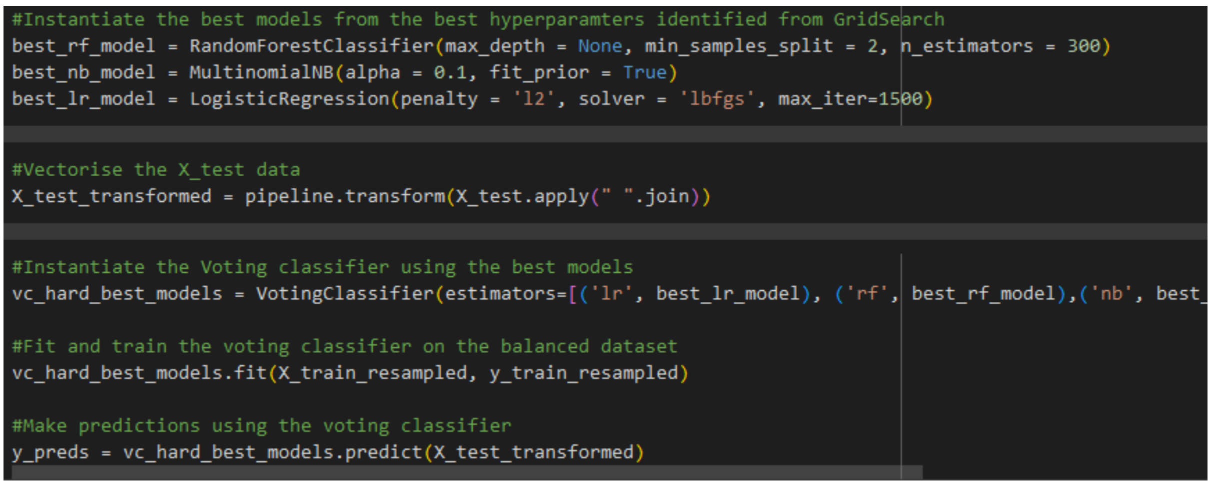
The best cross-validation accuracy was 94.7 %, but upon evaluating the model with the best parameters an accuracy score of only 77 % was reached.

4.2.4 Voting Classifiers

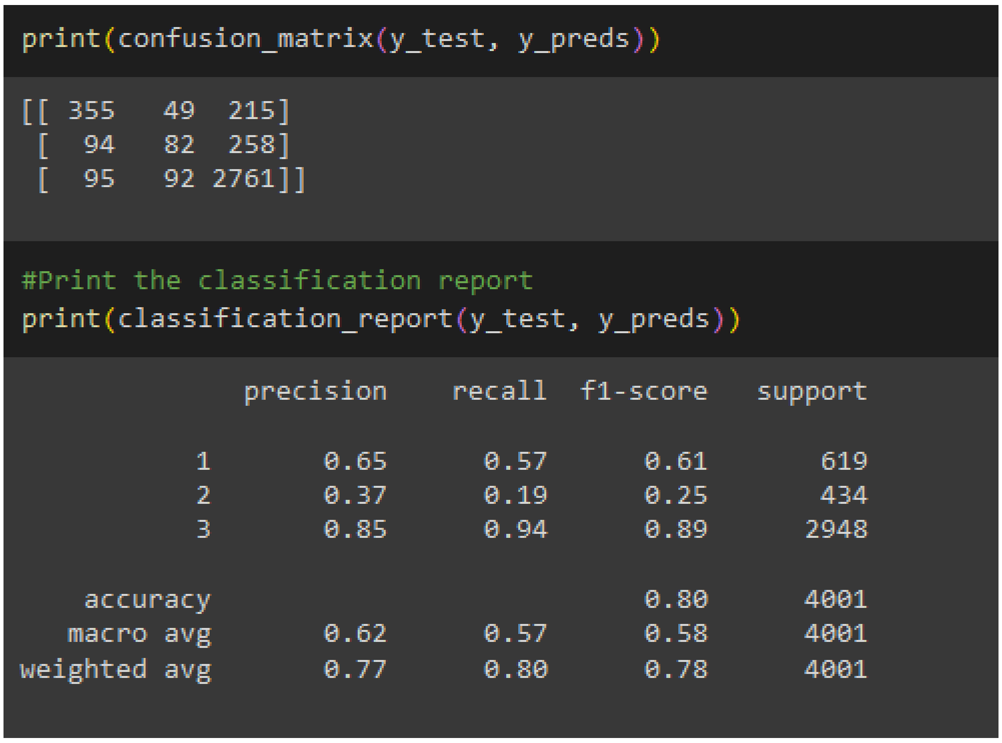
In this exercise, two voting classifier models were created and tested on the balanced training data. The first classifier was built from the models and best performing hyperparameters obtained in the previous subsection. The second classifier was created from 37 unique versions of the three classifiers, logistic regressor, random forest and multinomial Naïve Bayes.

4.2.4.1 Voting Classifier from best hyperparameters

The three classifiers and their best performing hyperparameters were instantiated. A voting classifier was then created incorporating these classifiers using the 'estimators' argument. The voting classifier was trained on the balanced 'X_train' and 'y_train' data, and predictions conducted on the vectorised 'X_test' data. The screenshots below show the code to train and test, as well as the results of the performance presented through a confusion matrix and a classification report.



```
#Instantiate the best models from the best hyperparamters identified from GridSearch
best_rf_model = RandomForestClassifier(max_depth = None, min_samples_split = 2, n_estimators = 300)
best_nb_model = MultinomialNB(alpha = 0.1, fit_prior = True)
best_lr_model = LogisticRegression(penalty = 'l2', solver = 'lbfgs', max_iter=1500)

#Vectorise the X_test data
X_test_transformed = pipeline.transform(X_test.apply(" ".join))

#Instantiate the Voting classifier using the best models
vc_hard_best_models = VotingClassifier(estimators=[('lr', best_lr_model), ('rf', best_rf_model),('nb', best_

#Fit and train the voting classifier on the balanced dataset
vc_hard_best_models.fit(X_train_resampled, y_train_resampled)

#Make predictions using the voting classifier
y_preds = vc_hard_best_models.predict(X_test_transformed)
```



```
print(confusion_matrix(y_test, y_preds))
```

```
[[ 355   49  215]
 [  94   82  258]
 [  95   92 2761]]
```

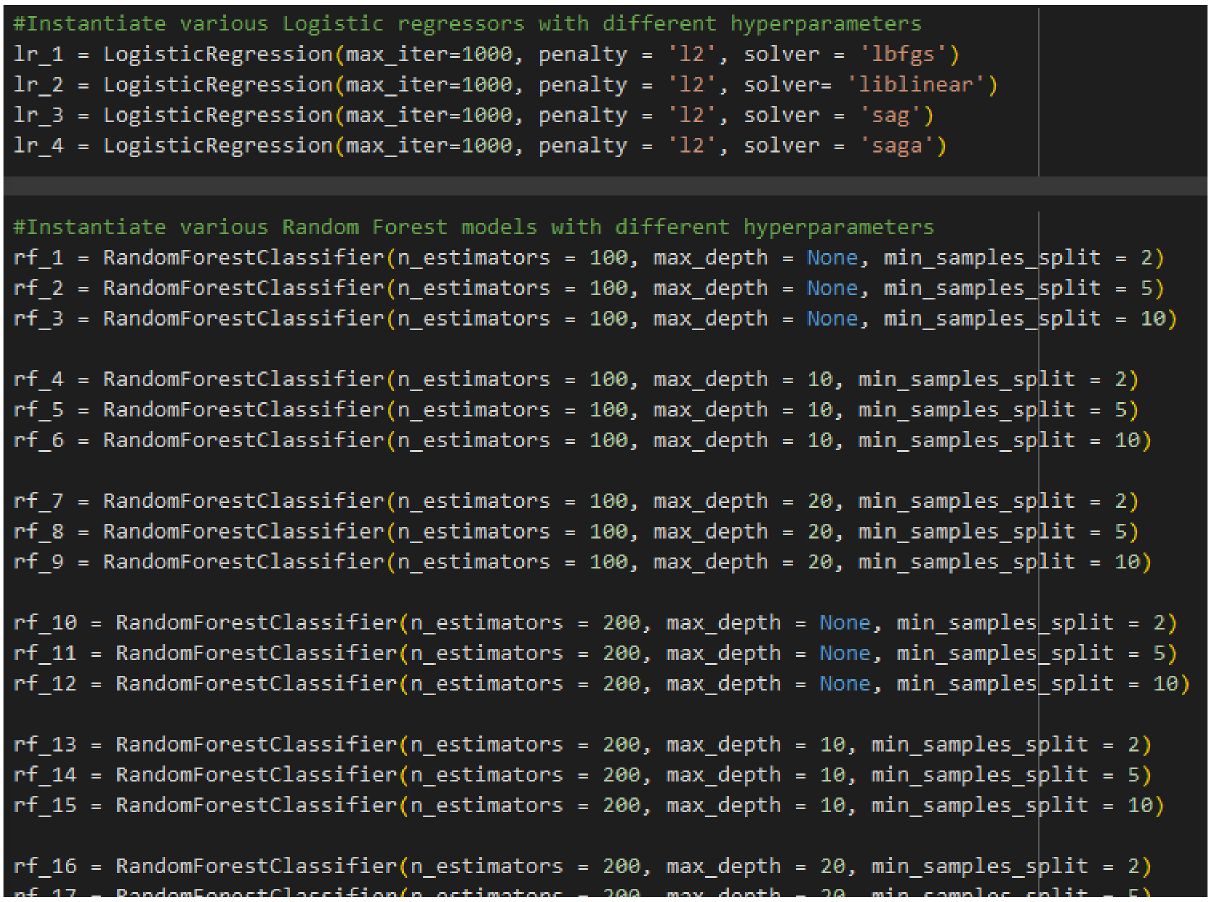
```
#Print the classification report
print(classification_report(y_test, y_preds))
```

```
              precision    recall  f1-score   support

           1       0.65      0.57      0.61       619
           2       0.37      0.19      0.25       434
           3       0.85      0.94      0.89      2948

    accuracy                           0.80      4001
   macro avg       0.62      0.57      0.58      4001
weighted avg       0.77      0.80      0.78      4001
```

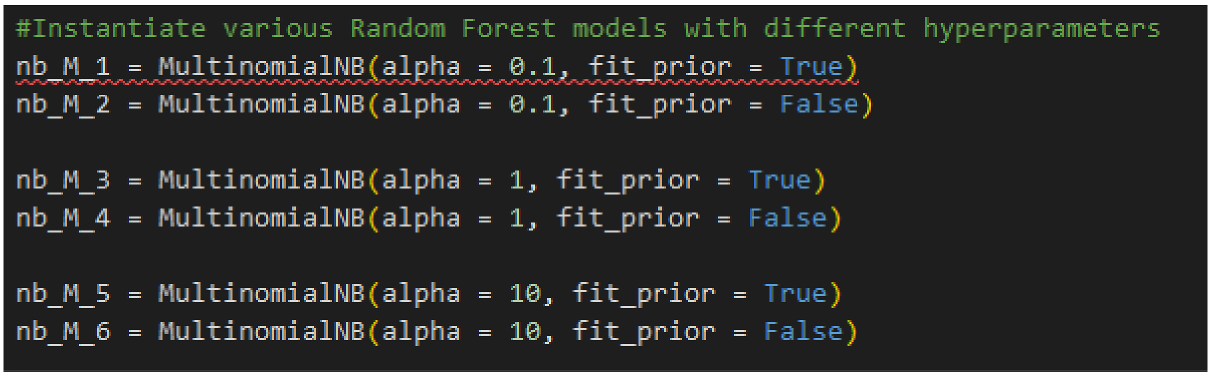
In this task class 1, 2, and 3 refer to bad, neutral and good reviews, respectively. The confusion matrix indicates some misclassifications, particularly between classes 1 and 3, where 215 instances of class 1 were misclassified as class 3. This suggests a potential bias in the model towards class 3 even after the training portion of the dataset was balanced using the SMOTE function. Class 3 has high precision and recall, indicating good performance for this class. However, class 2 has notably low precision (0.37) and recall (0.19), which implies that the model struggles to correctly identify and classify instances of class 2. The F1-score for class 2 is also low (0.25), reflecting the poor precision and recall. The F1-score for class 1 (0.61) and class 3 (0.89) are better, with class 3 being the highest, which aligns with its high precision and recall. While the overall accuracy is 0.80, the macro and weighted averages reveal disparities among the classes, suggesting that the model's performance is not consistent across different classes. The model may benefit from further tuning to address these imbalances.

4.2.4.2 Voting Classifier from various hyperparameters

The second voting classifier was built from numerous unique base model variations of the three classifiers encountered in the previous sections. The screenshot below shows a snippet of the instantiation of these models.

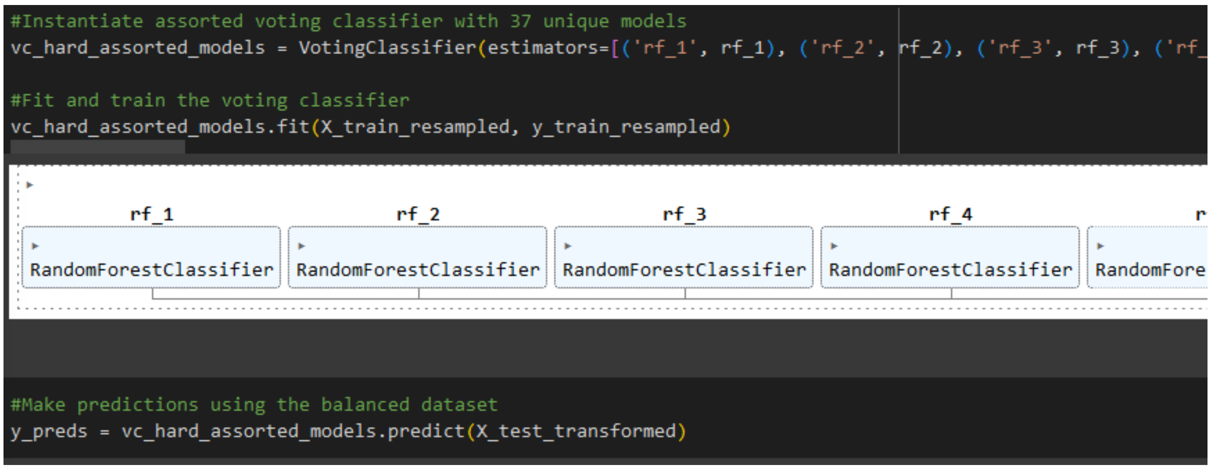


```
#Instantiate various Logistic regressors with different hyperparameters
lr_1 = LogisticRegression(max_iter=1000, penalty = 'l2', solver = 'lbfgs')
lr_2 = LogisticRegression(max_iter=1000, penalty = 'l2', solver= 'liblinear')
lr_3 = LogisticRegression(max_iter=1000, penalty = 'l2', solver = 'sag')
lr_4 = LogisticRegression(max_iter=1000, penalty = 'l2', solver = 'saga')

#Instantiate various Random Forest models with different hyperparameters
rf_1 = RandomForestClassifier(n_estimators = 100, max_depth = None, min_samples_split = 2)
rf_2 = RandomForestClassifier(n_estimators = 100, max_depth = None, min_samples_split = 5)
rf_3 = RandomForestClassifier(n_estimators = 100, max_depth = None, min_samples_split = 10)

rf_4 = RandomForestClassifier(n_estimators = 100, max_depth = 10, min_samples_split = 2)
rf_5 = RandomForestClassifier(n_estimators = 100, max_depth = 10, min_samples_split = 5)
rf_6 = RandomForestClassifier(n_estimators = 100, max_depth = 10, min_samples_split = 10)

rf_7 = RandomForestClassifier(n_estimators = 100, max_depth = 20, min_samples_split = 2)
rf_8 = RandomForestClassifier(n_estimators = 100, max_depth = 20, min_samples_split = 5)
rf_9 = RandomForestClassifier(n_estimators = 100, max_depth = 20, min_samples_split = 10)

rf_10 = RandomForestClassifier(n_estimators = 200, max_depth = None, min_samples_split = 2)
rf_11 = RandomForestClassifier(n_estimators = 200, max_depth = None, min_samples_split = 5)
rf_12 = RandomForestClassifier(n_estimators = 200, max_depth = None, min_samples_split = 10)

rf_13 = RandomForestClassifier(n_estimators = 200, max_depth = 10, min_samples_split = 2)
rf_14 = RandomForestClassifier(n_estimators = 200, max_depth = 10, min_samples_split = 5)
rf_15 = RandomForestClassifier(n_estimators = 200, max_depth = 10, min_samples_split = 10)

rf_16 = RandomForestClassifier(n_estimators = 200, max_depth = 20, min_samples_split = 2)
rf_17 = RandomForestClassifier(n_estimators = 200, max_depth = 20, min_samples_split = 5)
```



```
#Instantiate various Random Forest models with different hyperparameters
nb_M_1 = MultinomialNB(alpha = 0.1, fit_prior = True)
nb_M_2 = MultinomialNB(alpha = 0.1, fit_prior = False)

nb_M_3 = MultinomialNB(alpha = 1, fit_prior = True)
nb_M_4 = MultinomialNB(alpha = 1, fit_prior = False)

nb_M_5 = MultinomialNB(alpha = 10, fit_prior = True)
nb_M_6 = MultinomialNB(alpha = 10, fit_prior = False)
```

In total, 37 different models were assembled into the voting classifier and trained on the balanced training data as shown below.

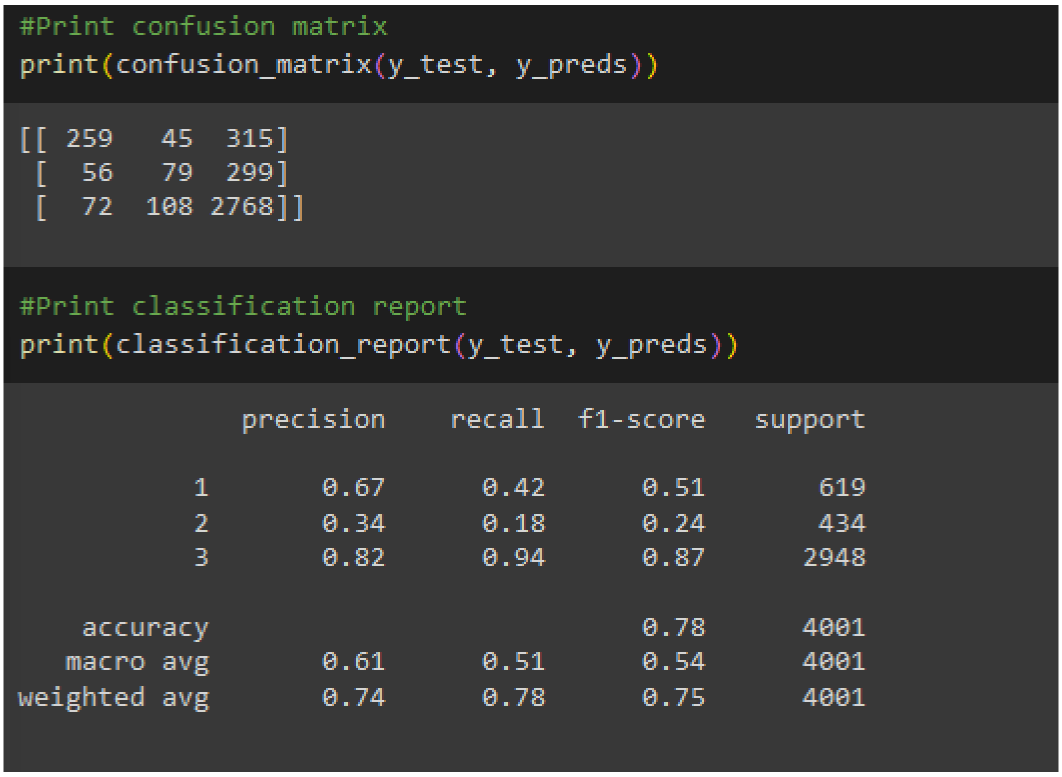


```
#Instantiate assorted voting classifier with 37 unique models
vc_hard_assorted_models = VotingClassifier(estimators=[('rf_1', rf_1), ('rf_2', rf_2), ('rf_3', rf_3), ('rf_

#Fit and train the voting classifier
vc_hard_assorted_models.fit(X_train_resampled, y_train_resampled)
```

```
      rf_1                      rf_2                      rf_3                      rf_4                      r
RandomForestClassifier    RandomForestClassifier    RandomForestClassifier    RandomForestClassifier    RandomFore
```

```
#Make predictions using the balanced dataset
y_preds = vc_hard_assorted_models.predict(X_test_transformed)
```

The evaluation of the classifier on the vectorised test data was done and the results are in the snapshot presented below.



```
#Print confusion matrix
print(confusion_matrix(y_test, y_preds))
```

```
[[ 259   45  315]
 [  56   79  299]
 [  72  108 2768]]
```

```
#Print classification report
print(classification_report(y_test, y_preds))
```

```
              precision    recall  f1-score   support

           1       0.67      0.42      0.51       619
           2       0.34      0.18      0.24       434
           3       0.82      0.94      0.87      2948

    accuracy                           0.78      4001
   macro avg       0.61      0.51      0.54      4001
weighted avg       0.74      0.78      0.75      4001
```

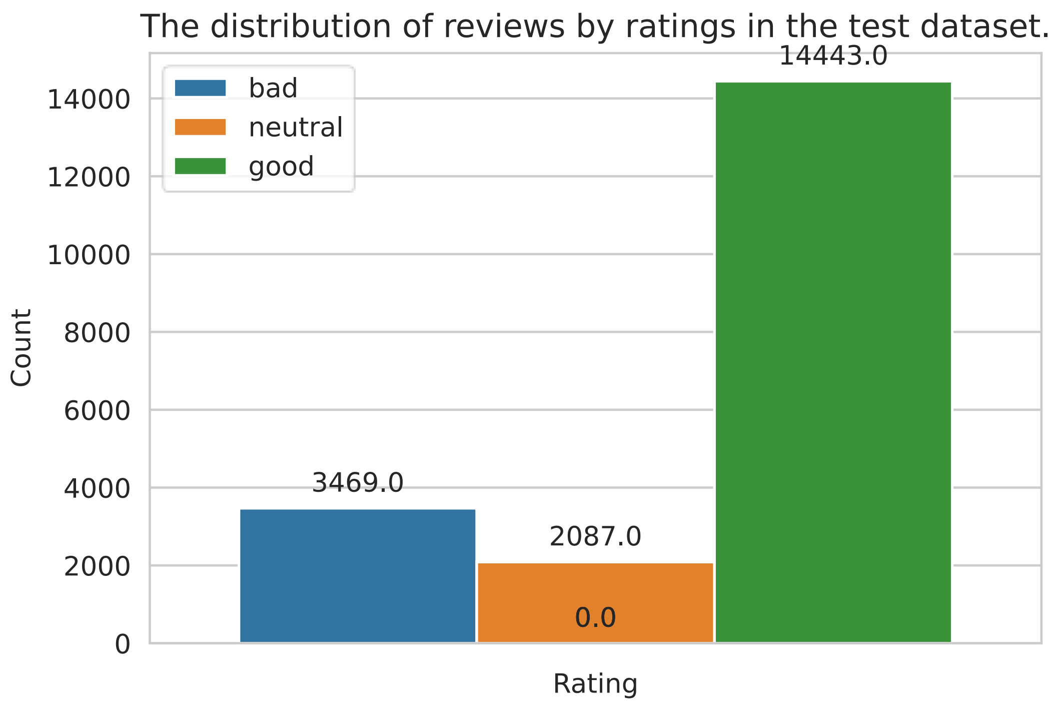
The confusion matrix shows some misclassification between classes 1 and 3, where 315 instances of class 1 were misclassified as class 3. This similarly to the first voting classifier, suggests the model may struggle to distinguish between these classes. Class 1 has a precision of 0.67 but a low recall of 0.42, indicating it is precise but not as sensitive. Class 2 has both low precision (0.34) and recall (0.18), which is concerning as it implies a high rate of both false positives and false negatives. The F1-scores, which balance precision and recall, are notably low for classes 1 and 2, at 0.51 and 0.24, respectively. This suggests the model's performance on these classes is not ideal. The support values show a significant imbalance in the dataset, with class 3 having many more instances than classes 1 and 2. This could bias the model's performance towards class 3. Overall, while the model shows decent accuracy and excellent performance for class 3, it appears to struggle with classes 1 and 2, which could be due to class imbalance or other factors affecting the model's ability to learn the distinctions between these classes. Improvements might be needed in data preprocessing, feature selection, or model parameters to enhance overall performance.

Comparing the two models' overall performance and the trade-offs, it is recommendable to choose the first voting classifier over the latter. The former has marginally better accuracy and F1-scores. Although the latter has slightly better precision for class 1, the former's better recall balances this out. The second voting classifier seems more robust across different metrics and shows slightly better generalisation.

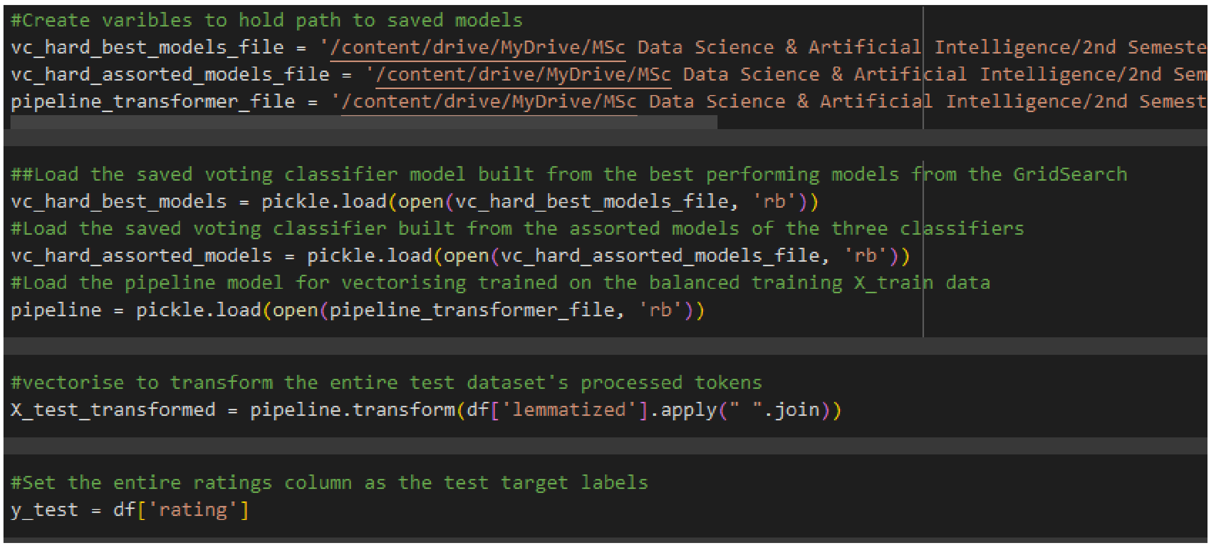
5. Analysis Results

A new jupyter notebook testing environment was created and the test dataset was loaded into a pandas dataframe called 'df' in a similar manner to the training dataset import. The reviews text data was also processed using the same steps from tokenisation to lemmatisation. The distribution of all 19 999 reviews in the test dataset is shown in Figure 2 below.

Figure 2: The distribution of reviews by ratings in the test dataset.

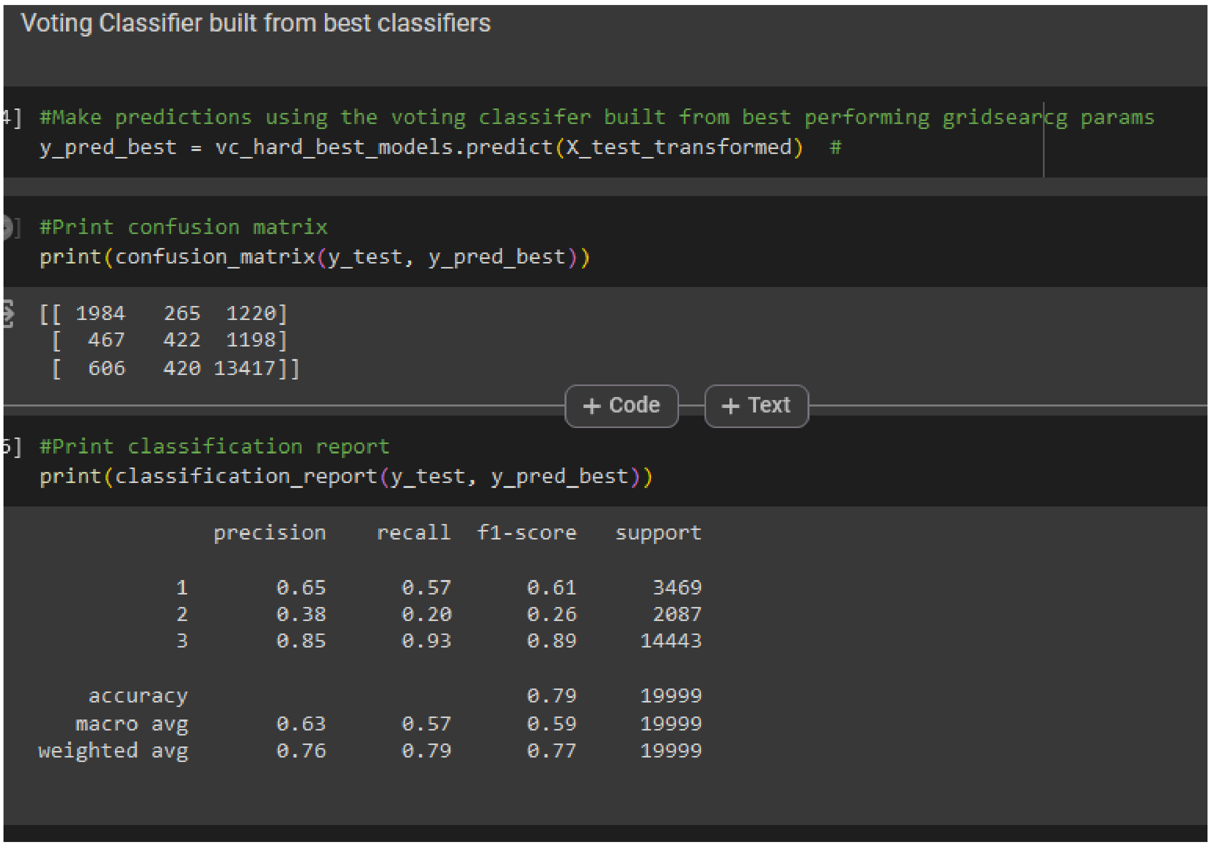


The test dataset shows an imbalance in class distribution of the same nature as that seen from the training dataset, with most instances having a good rating or class 3. The pickle.load() function was used to load the previously saved voting classifier and vectorisation pipeline models. The screenshot below shows the model reload onto new environment.



```
#Create varibles to hold path to saved models
vc_hard_best_models_file = '/content/drive/MyDrive/MSc Data Science & Artificial Intelligence/2nd Semeste
vc_hard_assorted_models_file = '/content/drive/MyDrive/MSc Data Science & Artificial Intelligence/2nd Sem
pipeline_transformer_file = '/content/drive/MyDrive/MSc Data Science & Artificial Intelligence/2nd Semest

##Load the saved voting classifier model built from the best performing models from the GridSearch
vc_hard_best_models = pickle.load(open(vc_hard_best_models_file, 'rb'))
#Load the saved voting classifier built from the assorted models of the three classifiers
vc_hard_assorted_models = pickle.load(open(vc_hard_assorted_models_file, 'rb'))
#Load the pipeline model for vectorising trained on the balanced training X_train data
pipeline = pickle.load(open(pipeline_transformer_file, 'rb'))

#vectorise to transform the entire test dataset's processed tokens
X_test_transformed = pipeline.transform(df['lemmatized'].apply(" ".join))

#Set the entire ratings column as the test target labels
y_test = df['rating']
```

The pipeline model previously trained was used to vectorise the processed string tokens into numerical data. The two voting classifier models were used to predict class labels for the entire test dataset. The results for the classifiers are shown in the screenshot below.


Voting Classifier built from best classifiers

```
#Make predictions using the voting classifer built from best performing gridsearcg params
y_pred_best = vc_hard_best_models.predict(X_test_transformed)  #
```

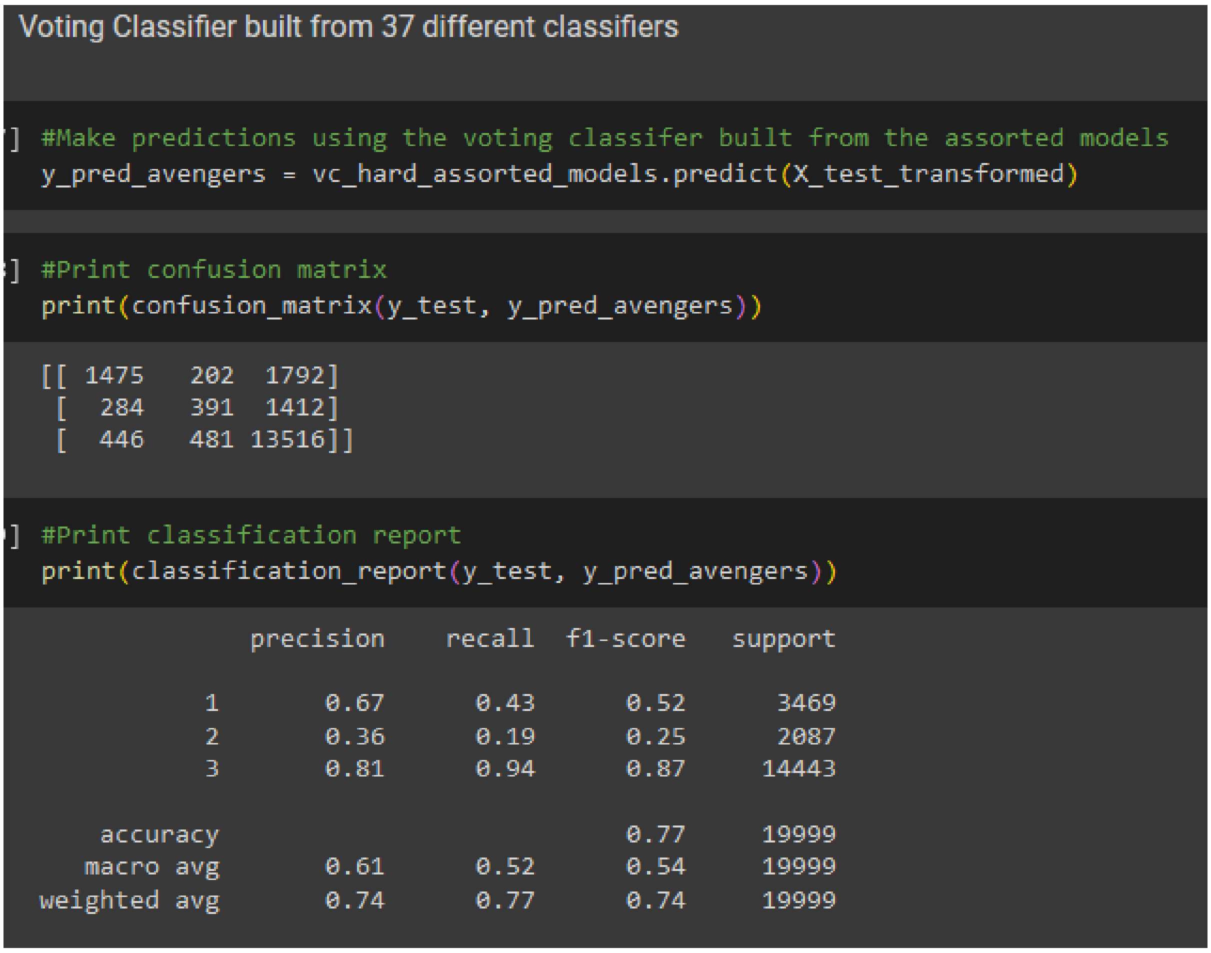
```
#Print confusion matrix
print(confusion_matrix(y_test, y_pred_best))
```

```
[[ 1984   265  1220]
 [  467   422  1198]
 [  606   420 13417]]
```

+ Code + Text

```
#Print classification report
print(classification_report(y_test, y_pred_best))
```

```
              precision    recall  f1-score   support

           1       0.65      0.57      0.61      3469
           2       0.38      0.20      0.26      2087
           3       0.85      0.93      0.89     14443

    accuracy                           0.79     19999
   macro avg       0.63      0.57      0.59     19999
weighted avg       0.76      0.79      0.77     19999
```


Voting Classifier built from 37 different classifiers

```
#Make predictions using the voting classifer built from the assorted models
y_pred_avengers = vc_hard_assorted_models.predict(X_test_transformed)
```

```
#Print confusion matrix
print(confusion_matrix(y_test, y_pred_avengers))
```

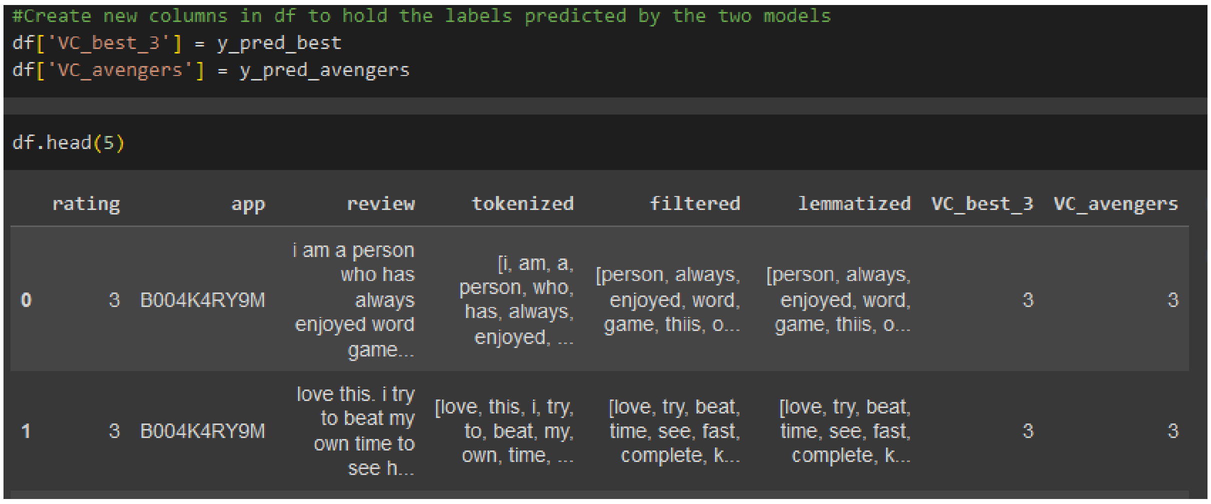
```
[[ 1475   202  1792]
 [  284   391  1412]
 [  446   481 13516]]
```

```
#Print classification report
print(classification_report(y_test, y_pred_avengers))
```

```
              precision    recall  f1-score   support

           1       0.67      0.43      0.52      3469
           2       0.36      0.19      0.25      2087
           3       0.81      0.94      0.87     14443

    accuracy                           0.77     19999
   macro avg       0.61      0.52      0.54     19999
weighted avg       0.74      0.77      0.74     19999
```

The voting classifier with best hyperparameter models had the higher accuracy of 0.79, performing better overall compared to the assorted voting classifier. It had a higher recall for Class 3 (0.93) indicating it correctly identifies this class more often. The precision for Class 1 was 0.65, and the F1-score across classes shows a balanced performance with a macro average of **0**.59. The latter classifier had the lower accuracy of 0.77, indicating it makes more mistakes overall. The recall for Class 1 was only 0.43, showing it struggles to identify this class correctly. The precision for Class 1 was 0.67, slightly better than former classifier, but the macro average F1-score was lower at 0.54. The former is more dependable, especially in identifying Class 3, which has the largest support. The latter, while slightly better at precision for Class 1, falls behind in overall accuracy and recall for Class 1. Both classifiers like in the training exercises, show room for improvement in Class 2 performance.

The class predictions for the voting classifier with best hyperparameter models were added into a new column in the test dataframe 'df'.

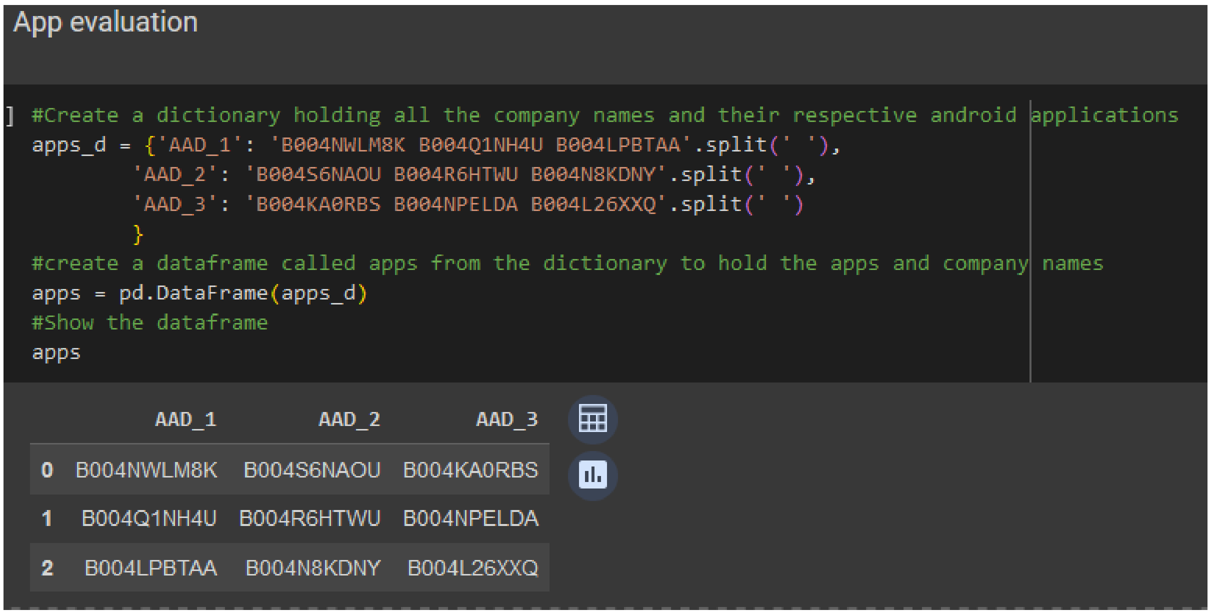


```
#Create new columns in df to hold the labels predicted by the two models
df['VC_best_3'] = y_pred_best
df['VC_avengers'] = y_pred_avengers
```

```
df.head(5)
```

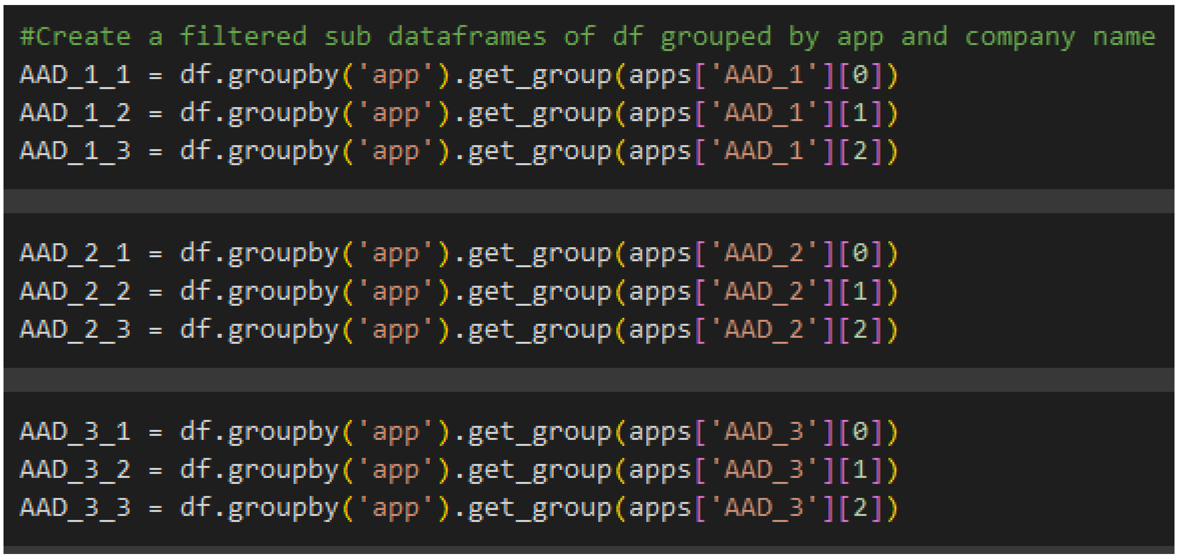
	rating	app	review	tokenized	filtered	lemmatized	VC_best_3	VC_avengers
0	3	B004K4RY9M	i am a person who has always enjoyed word game...	[i, am, a, person, who, has, always, enjoyed, ...	[person, always, enjoyed, word, game, thiis, o...	[person, always, enjoyed, word, game, thiis, o...	3	3
1	3	B004K4RY9M	love this. i try to beat my own time to see h...	[love, this, i, try, to, beat, my, own, time, ...	[love, try, beat, time, see, fast, complete, k...	[love, try, beat, time, see, fast, complete, k...	3	3

A new dataframe called 'apps' was created to hold the android application codes under the respective company names. This dataframe was used to query for sub-dataframes in 'df' containing instances grouped by each android application as shown below.


App evaluation

```
] #Create a dictionary holding all the company names and their respective android applications
  apps_d = {'AAD_1': 'B004NWLM8K B004Q1NH4U B004LPBTAA'.split(' '),
          'AAD_2': 'B004S6NAOU B004R6HTWU B004N8KDNY'.split(' '),
          'AAD_3': 'B004KA0RBS B004NPELDA B004L26XXQ'.split(' ')
          }
  #create a dataframe called apps from the dictionary to hold the apps and company names
  apps = pd.DataFrame(apps_d)
  #Show the dataframe
  apps
```

	AAD_1	AAD_2	AAD_3
0	B004NWLM8K	B004S6NAOU	B004KA0RBS
1	B004Q1NH4U	B004R6HTWU	B004NPELDA
2	B004LPBTAA	B004N8KDNY	B004L26XXQ



```
#Create a filtered sub dataframes of df grouped by app and company name
AAD_1_1 = df.groupby('app').get_group(apps['AAD_1'][0])
AAD_1_2 = df.groupby('app').get_group(apps['AAD_1'][1])
AAD_1_3 = df.groupby('app').get_group(apps['AAD_1'][2])

AAD_2_1 = df.groupby('app').get_group(apps['AAD_2'][0])
AAD_2_2 = df.groupby('app').get_group(apps['AAD_2'][1])
AAD_2_3 = df.groupby('app').get_group(apps['AAD_2'][2])

AAD_3_1 = df.groupby('app').get_group(apps['AAD_3'][0])
AAD_3_2 = df.groupby('app').get_group(apps['AAD_3'][1])
AAD_3_3 = df.groupby('app').get_group(apps['AAD_3'][2])
```

The sub-dataframes were then used to create bar plots shown in figures 3 and 4.

Figure 3: Bar plots showing each application's review classification distribution according to the best voting classifier's predictions.

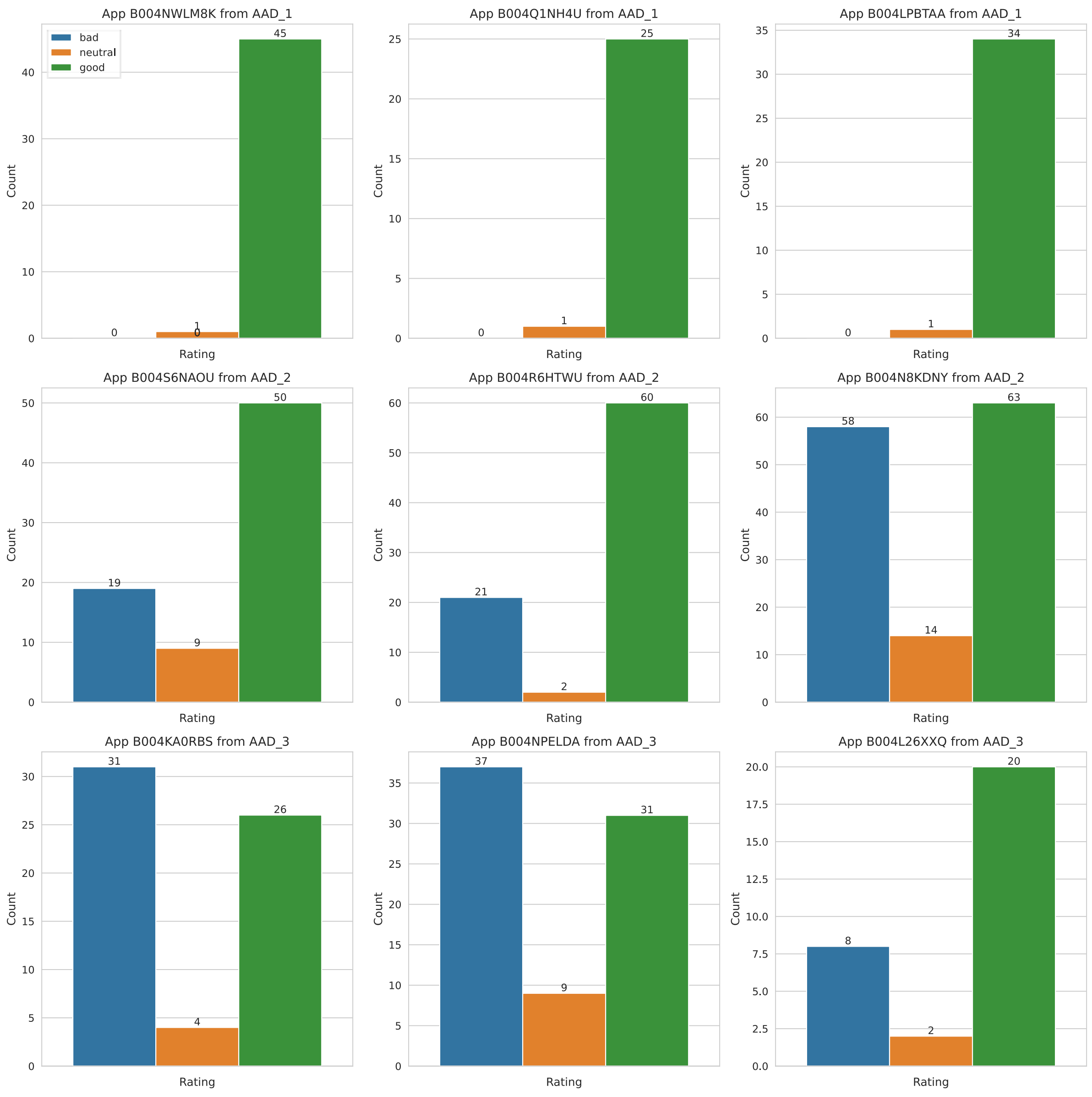


FIgure 4: Barplot showing percentage of good reviews out of total reviews for each app from the three companies.

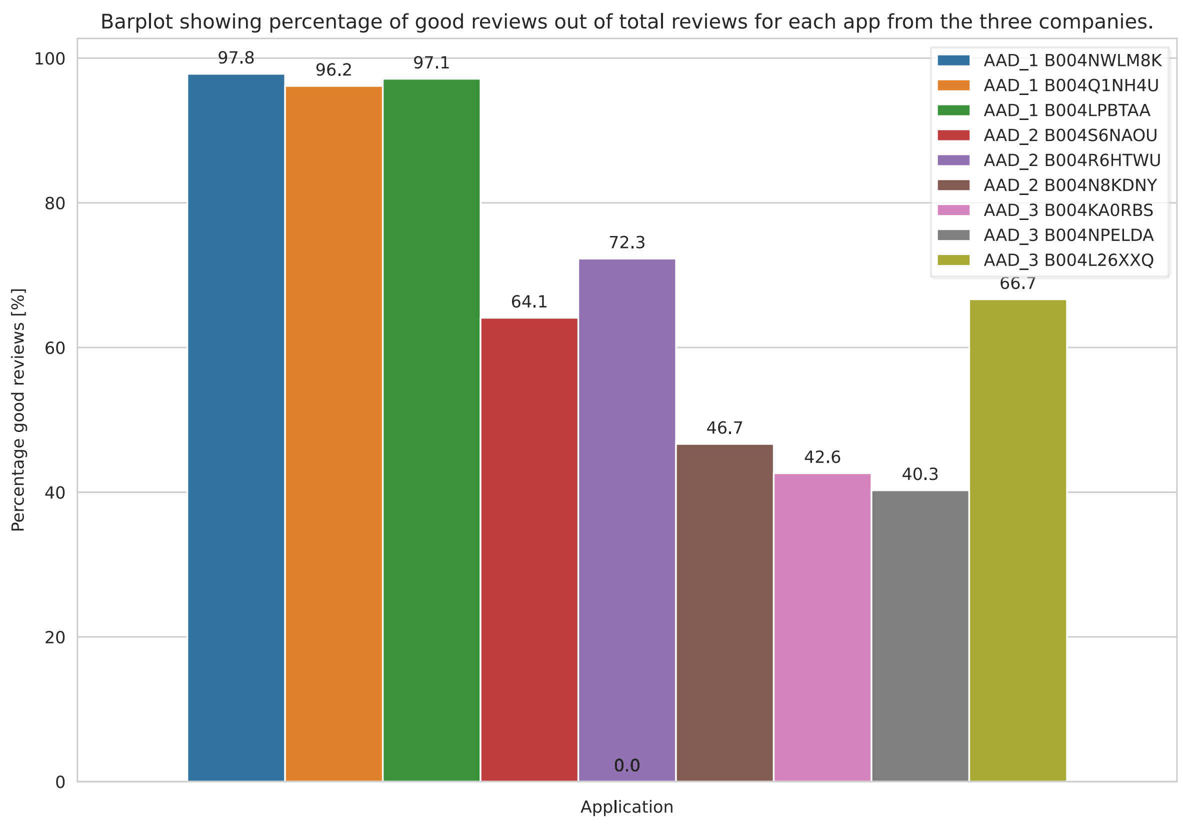


Figure 3 shows that company AAD_1 had the overall best rated application out of the 3 companies. Figure 4 confirms this conclusion as it shows that the application with code 'B004NWLM8K' had 97.8 % of its reviews rated good. The worst rated application by percentage of good reviews was 'B004NPELDA'.

However, the metrics are not fully understood without considering the total number of reviews each application got. This is shown in Figure 5.

Figure 5: Barplot showing total number of reviews for each application from the three companies.

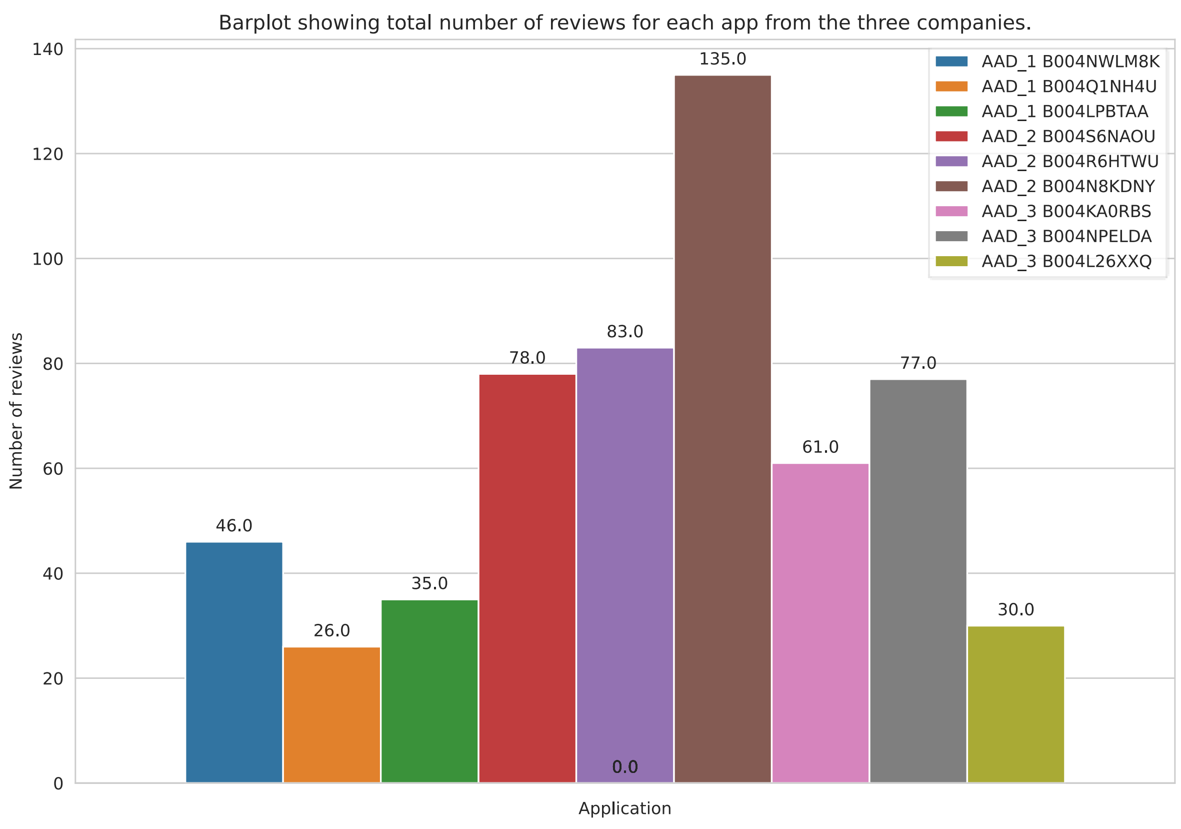


Figure 5 shows that application 'B004N8KDNY' had the most reviews out of the 9 apps. The application also has the most good reviews by count, higher than the app 'B004NWLM8K'. This presents a trade-off on the decision for the investment allocation. App 'B004NWLM8K' has the best percentage of good reviews out of the lot but has one third the number of total reviews of app 'B004N8KDNY'. Furthermore, app 'B004N8KDNY' has the most positive reviews by raw count albeit also having the greatest number of bad reviews according to Figure 3. This implies that for app 'B004N8KDNY' customers either like it or don't in majority of the cases. The higher total number of reviews hints that 'B004N8KDNY' is more popular than 'B004NWLM8K' even though the few who download the latter like it.

The recommendation for investment leans more on the popular 'B004N8KDNY' with the most positive reviews, however, further sentiment analysis of the bad reviews will have to be conducted to identify what features of the app people don't like. If the unfavourable features are issues that can be address and aren't fundamental to the app's very essence, then the investment can be placed on it. The decision entirely depends on whether the app can be modified in the future by developers to address issues. Otherwise, 'B004NWLM8K' will be a good alternative. Its mediocre popularity, assumed from its low number of reviews is the only concern. Perhaps an investigation into how long the application has been in the market could further shed light on its validity for investment choice. In conclusion app 'B004N8KDNY' form AAD_2 is favoured for investment.

6. Discussion

The exercises to find the best android application for investment allocation were conducted leveraging sentiment analysis of Amazon reviews. Reports of varying levels of accuracy across different machine learning models were obtained. For instance, logistic regression showed a high cross-validation accuracy but a lower performance on the training dataset. This discrepancy could stem from overfitting during cross-validation or a mismatch between training and validation data distributions. To improve results, a more rigorous validation strategy, such as k-fold cross-validation, could be implemented to ensure the model's generalisability.

The experiments indicated a struggle with classifying neutral reviews (class 2), which could be due to class imbalance. Even after employing SMOTE to balance the dataset, the model still showed bias towards positive reviews (class 3). Future work could explore alternative oversampling techniques or cost-sensitive learning to better handle class imbalance.

The use of 'CountVectorizer' and 'TF-IDF' may have led to a loss of semantic meaning, as these methods do not account for word order or context. Advanced techniques like word embeddings or deep learning models could be employed to capture more nuanced text features, potentially improving sentiment classification accuracy.

The exercise involved experimenting with voting classifiers, combining multiple models to enhance prediction accuracy. However, the complexity of these ensemble models can lead to computational challenges and difficulties in interpretation. Simplifying the ensemble or using more interpretable base models could make the results more transparent and easier to analyse.

Overall, the task completion presents a comprehensive analysis of sentiment classification using various machine learning techniques. However, addressing the mentioned issues could lead to more robust and reliable results in future studies.

BERT, a transformer-based model, has revolutionised NLP tasks, including sentiment analysis. It captures contextual information by considering both left and right context words. Fine-tuning BERT on sentiment-specific datasets can yield impressive results. Although GPT models are primarily designed for text generation, they can also be fine-tuned for sentiment analysis. Their ability to understand context and generate coherent text makes them valuable for this task. XLNet, an extension of BERT, addresses its limitations by considering all permutations of words in a sentence. It achieves state-of-the-art performance on various NLP benchmarks, including sentiment analysis. These are just a few of the more advanced current models used in NLP that could be attempted to improve upon the results from this iteration of the task.

7. Conclusions

provide a summary of the aims of the project and how well you've achieved them.

8. References

BANDI, R., SAI, M., RAJAVARDHAN SR., SATHWIK RB. and VEMPATI SV., 2023. Voting Classifier-based crop recommendation. *SN Computer Science*. 4(5). Available from: https://doi.org/10.1007/s42979-023-01995-8 [Accessed 23 March 2024].

BASA, SN. and BASARSLAN, MS., 2023. Sentiment analysis using machine learning techniques on IMDB dataset. *2023 7th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. Available from: https://doi.org/10.1109/ismsit58785.2023.10304923 [Accessed 23 March 2024].

BIRUNTHA, S., ARUL, GS. and ASHWIN, B., 2022. Distinguishing reviews through sentiment analysis using machine learning techniques. *2022 8th International Conference on Advanced Computing and Communication Systems (ICACCS)*. Available from: https://doi.org/10.1109/icaccs54159.2022.9785354 [Accessed 24 March 2024].

BOSE, S. and ROY, R., 2023. Enhancing fake news detection with sentiment analysis using machine learning. *2023 7th International Conference on Electronics, Materials Engineering & Nano-Technology (Imtech)*. Available from: https://doi.org/10.1109/iementech60402.2023.10423496 [Accessed 22 March 2024].

CHEN, Q., ZHANG, ZL., HUANG, WP., WU, J. and LUO, XG., 2022. PF-SMOTE: A novel parameter-free SMOTE for imbalanced datasets. *Neurocomputing*. 498, pp.75–88. Available from: https://doi.org/10.1016/j.neucom.2022.05.017 [Accessed 3 April 2024].

GANESAN, K., 2019. *How to use Tfidftransformer & Tfidfvectorizer - A short tutorial*. [online] Kavita Ganesan, PhD. Available from: https://kavita-ganesan.com/tfidftransformer-tfidfvectorizer-usage-differences/ [Accessed 3 April 2024].

HE, J. and HU, H., 2022. MF-BERT: Multimodal fusion in pre-trained BERT for sentiment analysis. *IEEE Signal Processing Letters*. 29, pp.454–458. Available from: https://doi.org/10.1109/lsp.2021.3139856 [Accessed 23 March 2024].

JEMAI, F., HAYOUNI, M. and BACCAR, S., 2021. Sentiment analysis using machine learning algorithms. *2021 International Wireless Communications and Mobile Computing (IWCMC)*. Available from: https://doi.org/10.1109/IWCMC51323.2021.9498965 [Accessed 1 March 2024].

PAVITHA, N., PUNGLIYA, V., RAUT, A., BHONSLE, R., PUROHIT, A., PATEL, A. and SHASHIDHAR, R., 2022. Movie Recommendation and Sentiment Analysis Using Machine Learning. *Global Transitions Proceedings*. Available from: https://doi.org/10.1016/j.gltp.2022.03.012 [Accessed 2 April 2024].

WU, J., ZHU, T., ZHU, J., LI, T. and WANG, C., 2022. A optimized BERT for multimodal sentiment analysis. *ACM Transactions on Multimedia Computing, Communications, and Applications*. Available from: https://doi.org/10.1145/3566126 [Accessed 16 April 2024].