NLP Coursework
Part A – Sentiment Analysis

Koner Kalkanel

Abstract

The applications of sentiment analysis are wide-ranging, from providing businesses with valuable insight into their success of their product, as well as how to increase customer satisfaction, to allowing governments to monitor public opinion on current affairs. In this assignment, we investigate multiple ways in which we can identify the sentiment (positive or negative) of a set of movie reviews from IMDb using different machine learning algorithms. The purpose of the assignment is to explore the NLP pipeline from start to finish, specifically regarding sentiment analysis, beginning with analysing the task and defining the problem, pre-processing the data. Then, we will explore the different ways in which we can generate features, before finally evaluating different models (naïve bayes, logistic regression, and support vector machines) on the dataset, comparing their ability to correctly classify the polarity of movie reviews. Lastly, we will consider any future work, discussing how the results could be improved.

1 Introduction and Motivation

Sentiment analysis refers to the study of opinions and attitudes towards an entity (Medhat, Hassan and Korashy, 2014), combining computer science and linguistics to classify the sentiment of a document, a media post or maybe a book review, for example. This classification may range from the analysis of a whole document to certain phrases or sentences, where we try to classify the sentiment of each sentence in a document (Wankhade, Rao and Kulkarni, 2022).

Classifying text by sentiment has a variety of applications today, such as gathering reviews and opinion on consumer products, as well as being used to gauge the reputation of a brand (Feldman, 2013). Due to opinions and emotions being central to most human behaviour, sentiment analysis is one of the most researched areas in natural language processing, important in recommendation systems and helping businesses make decisions and conduct market research (Baid, Gupta and Chaplot, 2017). Therefore, it is important that sentiment analysis tasks are carried out accurately so that any decision made using sentiment analysis is based on the true feelings of the public and, from a business point of view, companies are trying to increase customer satisfaction based on accurate feelings towards their current products.

There are difficulties, however, with ensuring the sentiment of the text is classified accurately. Difficult problems for machine learning algorithms are presented by sarcasm and irony, where the sentiment may be the opposite of the words used to express an opinion (Wankhade, Rao and Kulkarni, 2022). The best way to handle satirical texts and humour is still a challenging problem today, and many choose to simply ignore dealing with it. Moreover, grammatical errors, spelling mistakes, and incorrect punctuation are also common problems that need to be considered, especially if the data is collected from social media, where users are very casual in their writing style (Shahnawaz and Astya, 2017; Wankhade, Rao and Kulkarni, 2022). Handling these mistakes is also challenging and is receiving much attention within the field of NLP (Li et al., 2022). As we will see, although such problems can affect the performance of a classifier, it is not necessarily detrimental if we do not handle such problems. It is important to remember, however, that this will depend on the context on the problem.

There are many different methods available to handle sentiment analysis problems, from how we pre-process our data and select our features, to the actual machine learning algorithm we use for sentiment analysis. Considering how we choose our features, we may take a statistical approach in selecting features for the problem, where we could filter our words common across all documents and measure, statistically, which words are more important for a document to have a positive sentiment or a negative sentiment (Birjali, Kasri and Beni-Hssane, 2021). We also have a variety of different machine learning approaches we can take regarding the algorithm used to classify a piece of text. For example, Naïve Bayes is a simple classifier that relies on Bayes Theorem to consider the probability that a given document is

from a certain class given its content, or we could use a support vector machine, that aims to find the optimal hyperplane that separates out data points of different classes (Birjali, Kasri and Beni-Hssane, 2021). Of course, we can also apply deep learning approaches, using deep neural networks (DNNs) or convolutional neural networks (CNNs) for example, for sentiment analysis, which is becoming increasingly popular due to their high performance (Yadav and Vishwakarma, 2019).

Furthermore, in this assignment we will test the performance of a Naïve Bayes classifier, a Support Vector Machine (SVM), and a logistic regression classifier on classifying whether a movie review is positive or negative, being trained on three different feature sets. The purpose is to explore how each model performs with respect to different features provided. Specifically, the three feature sets will only differ in their use of n-grams, so we can explore how being able to capture, bi-grams, for example may help improve performance. We will evaluate each classifier in terms of accuracy, precision and f1 score, as well as look at the ways in which we can change the hyperparameters for the SVM and logistic regression classifier to improve performance. We will see that, after tuning some hyperparameters, the logistic model performs the best, with an accuracy and F1 score of 87%. However, we will also see that no classifier performs significantly worse than the others, with all classifiers being able to achieve an accuracy and F1 score of at least 80%.

2 Related Work

Many different methods exist for sentiment analysis. Birjali, Kasri and Beni-Hssane (2021) explore these methods thoroughly. Methods between contexts begin to largely differ starting from the way in which features are selected. The simplest features included the presence of a term or its frequency, where we can apply TF-IDF to measure the importance of the term in deciding if a piece of text is positive or negative. Despite simplicity, this has still been found as an effective way to select features (Ahuja et al., 2019). Parts-of-Speech (PoS) tags can also be effective features. For instance, adjectives have been shown to be good indicators of sentiment (P. Tanawongsuwan, 2010).

After features have been selected, there is also many choices concerning which algorithm we can choose. For instance, SVMs are popular, aiming to create a decision boundary between data points of different sentiments (Moraes, Valiati and Gavião Neto, 2013). Probabilistic models are also used, with Naïve Bayes being one of the most commonly used algorithms. Despite its simplicity, many find that Naïve Bayes is an effective and quick approach (Birjali, Kasri and Beni-Hssane, 2021). Moreover, although supervised algorithms are favoured, unsupervised clustering algorithms can also be used, with Ma, Yuan, and Wu (2017) showing that K-means clustering is an effective technique for sentiment analysis, specifically performing well on balanced review datasets. Recently, deep learning approaches, although not explored in this assignment, are becoming more popular. For example, many are looking to use deep learning for better detection of sarcasm and irony within text (Yadav and Vishwakarma, 2019).

3 Experiment & Results

Now, we can look into the experiments I conducted concerning three typical classification models: Naïve Bayes, SVMs, and logistic regression. We will talk about these experiments regarding the typical NLP pipeline, starting with a quick overview of the problem. The task at hand is sentiment analysis, where we want to identify whether a movie review is either positive or negative. This can be seen as a binary classification task. The dataset supplied is 2000 positive reviews and 2000 negative reviews from IMDb. Following the NLP pipeline, we want to test and compare Naïve Bayes, SVM, and logistic regression classification on how well they are able to perform on this task.

3.1 Splitting Data into Training, Development and Test Set

After defining the problem, we then want to preprocess our data and split it into a training, development, and test set. The reivew data was initially read in as two dictionaries, with one for positive reviews and one for negative reviews, before being separated into a list of reviews and a list of corresponding labels. In doing so, I have treated the positive reviews as belonging to a class with label 1, and negative reviews as belonging to class 0.

```
import sklearn
from sklearn.model_selection import train_test_split

# We'll need to separate the content and the labels before using train_test_split.
reviews = [review for (_,review) in all_reviews]
labels = [label for (label,_) in all_reviews]


# We can split the data now into two parts, training and test data. Random_state here ensures reproducability.
X_train, X_test, y_train, y_test = train_test_split(reviews, labels, test_size=0.2, random_state=42)

# We can split this again into a training and dev set.
X_train, X_dev, y_train, y_dev = train_test_split(X_train, y_train, test_size=0.20, random_state=42)
```

Figure 1. Splitting the dataset into a training, development and test set using

As we can see in Figure 1, I have used sklearn's built in train_test_split method to split the data into a training, development, and test set (scikit-learn, n.d). I have first used train_test_split to split the data into a training set that is 80% of the original data, and 20% is reserved for the test data. I have then used this function again to reserve 20% of the training data for the development set. In summary, of the 4000 data points, this leaves 64% for training (2560), 16% for development (640), and 20% for testing (800).

```
print("Training Set Shape:", len(X_train), len(y_train))
print("Development Set Shape", len(X_dev), len(y_dev))
print("Testing Set Shape:", len(X_test), len(y_test))

Training Set Shape: 2560 2560
Development Set Shape 640 640
Testing Set Shape: 800 800
```

Figure 2. Number of instances in each set

We also want to make sure that the proportion of negative and positive reviews in each set are approximately the same to avoid bias. The proportions found are shown below and are appropriately proportional across each dataset. As such, we can now look to inspect our training data and consider how we can extract features.

Category	Overall	Training Set	Development Set	Testing Set
1	0.500000	0.494922	0.485938	0.527500
0	0.500000	0.505078	0.514062	0.472500

Figure 3. Proportion of each class in each set, found using code from week 4's lab 'AuthorProfiling'.

3.2 Investigating the Data

After the data was split into the three sets, I then inspected the training set to gain insight into the data to help inform how I should generate features. As seen below, I found that 46.90% of all words in every review in the training data was covered by the top 50 words. Upon inspecting these words, they were found to either be stop words, such as 'a' and 'the', which do not carry a lot of meaning, or punctuation. This is shown below.

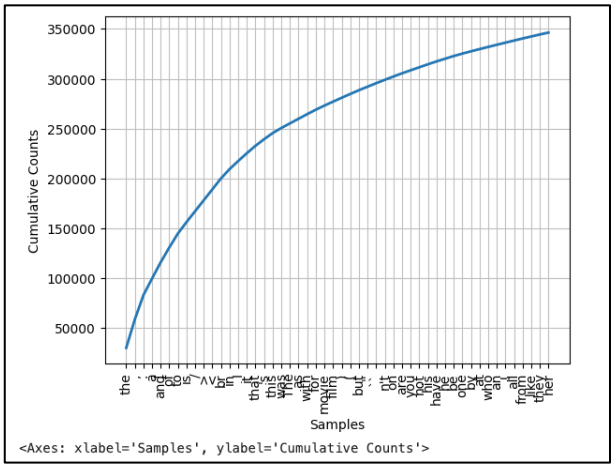


Figure 4. The top 50 words in the training data. The word counts are shown cumulatively.

```
top_50_words = word_freq_distribution.most_common(50) # Returns [(a,b)] where a is the word and b is the count
top_50_word_counts = sum(count for (word,count) in top_50_words)
total_word_count = len(all_words_across_reviews)
top_50_words_percentage = (top_50_word_counts/total_word_count)*100
print(f"Percentage of Words That Are Covered by the Top 50 Words: {top_50_words_percentage:.2f}%")

Percentage of Words That Are Covered by the Top 50 Words: 46.90%
```

Figure 5. The top 50 words were found to cover approximately 46.9% of all words in the training data.

Moreover, given that these words made up almost half the training data, and are also not informative as to whether a review is positive or negative, in generating features these types of words (stopwords), and punctuation, were always removed. To ensure that the same words with different capitalizations were not treated differently, every word was also lowercased.

```
def tokenize_and_lemmatize_document(document):
    stoplist = set(stopwords.words('english'))
    lem = WordNetLemmatizer()
    tokenized_word_list = [lem.lemmatize(word) for word in word_tokenize(document.lower())
                           if word.isalpha() and not word in stoplist and not word in string.punctuation]
    return tokenized_word_list
```

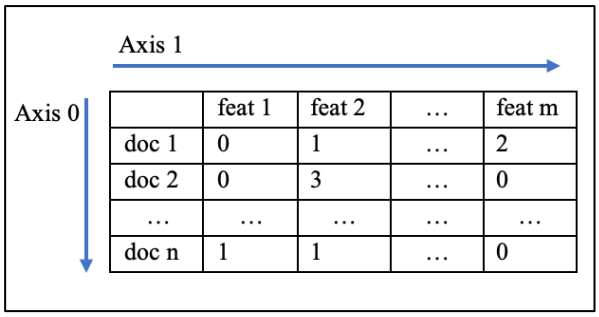
Figure 6. Tokenization and lemmatization method, given to sklearn's CountVecotrizer to ensure words are also lemmatized.

```
# Expects numpy array (document_term matrix) of the shape (x_samples, n_features)
def tfidf_normalization(document_term_matrix):
  # Calculating TF: tf(term) = count of term in a document / count of all terms in a document.
  document_term_counts = document_term_matrix.sum(axis=1)
  tf = document_term_matrix / document_term_counts[:, np.newaxis]

  # Calculating IDF: df(term) = log((no. of documents in dataseet / (no.of documents that contain the term+1))
  number_of_documents = document_term_matrix.shape[0]
  df = np.count_nonzero(document_term_matrix, axis=0) + 1
  idf = np.log(number_of_documents / df)

  tf_idf = tf * idf
  return tf_idf
```

Figure 7. Converting CountVectorizers matrix of term counts across documents to a matrix of tf-idf values.

3.3 Using CountVectorizer, Lemmatization and TF-IDF to Generate Features

In generating all features, sklearn's CountVectorizer (scikit-learn, n.d) was used to generate a matrix of term counts for each document. CountVectorizer can be supplied its own tokenizer method, and so I used this to also lemmatize the words, shown in figure 6. This method also handles tokenizing the words and removes any punctuation or stopwords. Given that CountVectorizer gives a matrix of term counts, this can then be used to create a matrix of TF-IDF values, which measures how important a term is in indicating the relevance, or in this case sentiment, of a document given that some terms are more likely to occur. This conversion to a TF-IDF matrix is shown in figure 7. We can first explore the intuition of the code.



	feat 1	feat 2	…	feat m
doc 1	0	1	…	2
doc 2	0	3	…	0
…	…	…	…	…
doc n	1	1	…	0

Figure 8. Image illustrating the matrix returned by CountVectorizer. Each element in the table represents the number of times a feature appears in a document.

Figure 8 illustrates an example of a matrix returned by CountVectorizer, where each element represents the number of times a feature (in our case these features will be terms) appears in each document. Then if we sum the counts along each column (axis 1 in figure 8) we can find how many terms appear in a specific document. This is stored as 'document_term_counts' in the code. Then, to calculate a normalised term frequency of a term in a document we simply need to divide each element along a row by the total number of terms in the document, which we calculated in 'document_term_counts.' This is represented as 'tf' in the code.

Similarly, to calculate the inverse document frequency we first need the number of total documents in our data set. This is represented by the number of rows in the document. The document frequency of each term can be found by counting the number of times a feature appears at least once across all documents. This is the same as counting the number of times we see an element bigger than zero along the rows for each column (axis 0 in figure 8). In the code this is represented by 'df'. We add one to every value to ensure no division of zero occurs when calculating inverse document frequency. We can then create a new matrix with one row and *m* columns, for *m* features, where each element represents the inverse document frequency of a term. Multiply this matrix with every row of our tf matrix previously

calculated will give us our tf-idf matrix for our terms. Figure 9 illustrates this point.

IDF Matrix ('idf' in code)

	feat 1	feat 2	…	feat m
Idf scores	$\log\frac{n}{df(feat\ 1)}$	$\log\frac{n}{df(feat\ 2)}$	…	$\frac{n}{df(feat\ m)}$

TF Matrix ('tf' in code)

	feat 1	feat 2	…	feat m
doc 1	$tf(feat\ 1, doc\ 1)$	$tf(feat\ 2, doc\ 1)$	…	$tf(feat\ m, doc\ 1)$
…	…	…	…	…
…	…	…	…	…
doc n	$tf(feat\ 1, doc\ n)$	$tf(feat\ 2, doc\ n)$	…	$tf(feat\ m, doc\ n)$

Figure 9. A visualization of our idf and tf matrix as calculated in the code. Multiplying each row in the TF matrix by the respective elements in the idf matrix will give us our idf-matrix.

3.4 Differentiating Features by N-Grams

All feature sets used in training were TF-IDF matrices. The difference, then, between each feature set was the use of n-grams. Specifically, feature set 1 only contained unigrams (words), feature set 2 contains unigrams and bigrams, while feature set 3 contained unigrams, bigrams, and trigrams. The generation of these three feature sets is seen below.

```
lemmatize_onegrams_vectorizer = CountVectorizer(tokenizer=tokenize_and_lemmatize_document)
feature_set_1_term_counts = lemmatize_onegrams_vectorizer.fit_transform(X_train)
feature_set_1 = tfidf_normalization(feature_set_1_term_counts.toarray())

lemmatize_twograms_vectorizer = CountVectorizer(tokenizer=tokenize_and_lemmatize_document,
                                                ngram_range=(1, 2))
feature_set_2_term_counts = lemmatize_twograms_vectorizer.fit_transform(X_train)
feature_set_2 = tfidf_normalization(feature_set_2_term_counts.toarray())

lemmatize_threegrams_vectorizer = CountVectorizer(tokenizer=tokenize_and_lemmatize_document,
                                                  ngram_range=(1,3))
feature_set_3_term_counts = lemmatize_threegrams_vectorizer.fit_transform(X_train)
feature_set_3 = tfidf_normalization(feature_set_3_term_counts.toarray())
```

Figure 10. Generating three sets of features, differing by size due to the range of n-grams considered.

In this way, our evaluation of these three sets for the will focus on the difference in performance based on how useful bigrams and unigrams are, for example, in trying to predict the sentiment of a movie review compared to only using unigrams. This is important since considering a wider range of n-grams requires significantly more features, and requires a lot more computational power, at the cost of possibly catching more context within a piece of text (Psomakelis et al., 2015). So, we can determine whether this computational cost, for this specific task, is worth it, given that we also want language models to be as fast as possible.

3.5 Personal Implementation of Naïve Bayes

First, I tested the feature sets using a personal implementation of Naïve Bayes. Naïve Bayes uses Bayes' Theorem to calculate the probability that a document is of a certain class given its content. Naïve Bayes solves the following given a movie document:

$$\begin{cases} P(content \mid positive)P(positive) \geq P(content \mid negative)P(negative) & \Rightarrow positive \\ otherwise, & \Rightarrow negative \end{cases}$$

Equation 1. The equation that naïve bayes attempts to solve in predicting whether a review is positive or negative.

First, to calculate P(positive) and P(negative) I use the following method.

```
def __calculate_prior_probabilities__(self, Y_train):
  training_data_size = len(Y_train)
  self.prior_probabilities[0] = np.log(np.count_nonzero(Y_train == 0) / training_data_size)
  self.prior_probabilities[1] = np.log(np.count_nonzero(Y_train == 1) / training_data_size)
```

Figure 11. Calculating the probability that a class is positive or negative, known as prior probabilities.

Note that I am using log probabilities, as otherwise these probabilities would be incredibly small, and approach a number that often cannot be represented by the computer, e.g., there are not enough bytes available in python to store a small enough decimal. By using log probabilities, we avoid such issues. Nevertheless, calculating these probabilities, known as prior probabilities, is relatively simple, as we can use the idea that:

$$P(positive) = \frac{No.\ positive\ reviews\ in\ the\ class}{No.\ all\ reviews\ in\ the\ class}$$

Calculating the feature likelihoods is the next important method, shown below.

```
def __calculate_feature_likelihoods__(self, X_train, Y_train, alpha):
  num_of_features = X_train.shape[1]

  # Shape of (2, num_of_features), with self.feature_likelihoods[i][j] representing p(feature j | class i)
  self.feature_likelihoods = np.zeros((len(self.class_labels), num_of_features))

  for class_label in self.class_labels:
    instances_of_class_label = X_train[Y_train == class_label]
    feature_counts_along_documents = np.sum(instances_of_class_label, axis=0)  + alpha
    total_number_of_features = np.sum(instances_of_class_label) + (alpha)*(num_of_features)
    self.feature_likelihoods[class_label, :] = np.log(feature_counts_along_documents / total_number_of_features)
```

Figure 12. Calculating the feature likelihoods, representing P(term|class)

This code makes use of the equation for $P(term\mid class)$ as:

$$\frac{No.of\ times\ \ this\ 'term'\ appears\ in\ the\ class}{No.of\ \ all\ terms\ that\ appear\ in\ the\ class}$$

In the code, we add an alpha value to account for times when a term does not appear in the class at all, ensuring this probability is not noted as zero. These probabilities are stored in a 2D list, where element [i][j] represents the probability that feature j appears in class i.

```python
# Class Description
  # A Naive Bayes Classifier implementation that assumes binary labels.
  # X_Train - a numpy array (document_term matrix) of the shape (n_samples, n_features)
  # Y_Train - a numpy array of shape (n_samples,), which correspond to the labels of the respective rows in X_train
  # X_Test  - a numpy array (document_term matrix) of the shape (x_samples, n_features)

class NaiveBayesClassifier:
  def __init__(self):
    self.class_labels = [0,1]
    self.prior_probabilities = [0,0]
    self.feature_likelihoods = None

  def train(self, X_train, Y_train, alpha=1):
    self.__calculate_prior_probabilities__(Y_train)
    self.__calculate_feature_likelihoods__(X_train, Y_train, alpha)

  def __calculate_prior_probabilities__(self, Y_train):
    training_data_size = len(Y_train)
    self.prior_probabilities[0] = np.log(np.count_nonzero(Y_train == 0) / training_data_size)
    self.prior_probabilities[1] = np.log(np.count_nonzero(Y_train == 1) / training_data_size)

  def __calculate_feature_likelihoods__(self, X_train, Y_train, alpha):
    num_of_features = X_train.shape[1]

    # Shape of (2, num_of_features), with self.feature_likelihoods[i][j] representing p(feature j | class i)
    self.feature_likelihoods = np.zeros((len(self.class_labels), num_of_features))

    for class_label in self.class_labels:
      instances_of_class_label = X_train[Y_train == class_label]
      feature_counts_along_documents = np.sum(instances_of_class_label, axis=0)  + alpha
      total_number_of_features = np.sum(instances_of_class_label) + (alpha)*(num_of_features)
      self.feature_likelihoods[class_label, :] = np.log(feature_counts_along_documents / total_number_of_features)

  def predict(self, X_test):
    predictions = []
    for document in X_test:
      class_probabilities = [self.prior_probabilities[0], self.prior_probabilities[1]]
      for class_label in self.class_labels:
        for feature_index, feature_count in enumerate(document):
          class_probabilities[class_label] += feature_count * self.feature_likelihoods[class_label, feature_index]
      predictions.append(np.argmax(class_probabilities))
    return predictions
```

Figure 13. My implementation of a naïve bayes classifier.

The entire code for the classifier is shown in figure 13, which includes the last important method: the prediction method, which expects a matrix in the same form of the feature set, which in our case will be a matrix of tf-idf values, and calculates the probability that a review is of a certain task using the Naïve Bayes assumption, where we assume that the terms are independent of each other in calculating P(content | class). For example,

$$P(\text{I love this movie just kidding} \mid \text{positive review})$$

can be calculated as:

$$P(\text{I} \mid \text{positive review}) \times P(love \mid \text{positive review}) \times \ldots \times P(kidding \mid positive\ review)$$

After the prediction method has calculated whether it is more likely to see this content given the class is negative or positive for all reviews, it returns the predicted labels.

3.6 Performance of Naïve Bayes Classifier

Now, we can look at the performance of the Naïve Bayes classifier on the three test sets. Figure 14 shows the performance of the classifier on feature set 1 as an example. Note how the classifier has only been evaluated on the development set, which has been transformed using the exact same functions that were used to create the first feature set. This is to ensure that we are testing on a development set that has the exact same features as our training set.

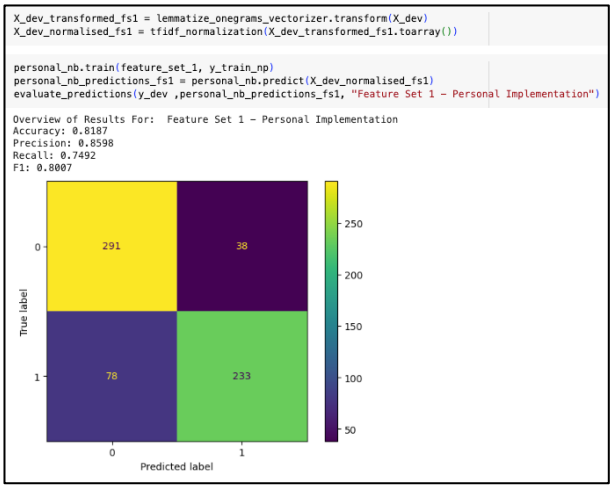


Figure 14. Performance of personal naïve bayes implementation on the first feature set.

Testing this on sklearn's implementation of the multinomial Naïve Bayes classifier, I found the exact same performance, indicating that my Naïve Bayes classifier was a correct implementation.

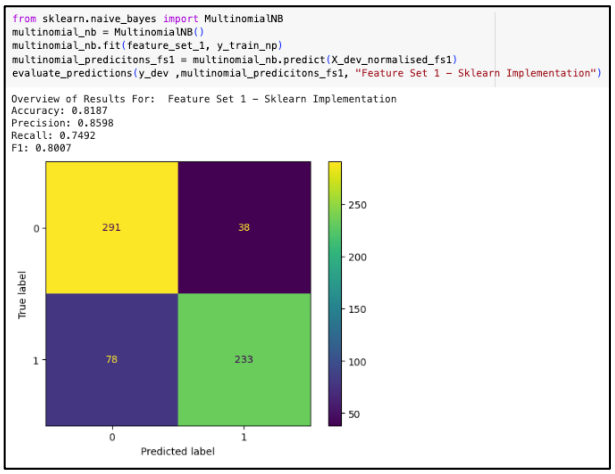


Figure 15. Performance of sklearn's multinomial naïve bayes on feature set 1.

The results have been summarised in the following table, with the results of the sklearn's multinomial naïve bayes giving the exact same scores for each metric.

Results of Naïve Bayes Classifier					
Feature Set	N-Grams Contained in Set	Precision	Recall	F1	Accuracy
1	Unigrams	0.8598	0.7492	0.8007	0.8187
2	Unigrams, Bigrams	0.8902	0.7299	0.8021	0.8250
3	Unigrams, Bigrams, Trigrams	0.8921	0.6913	0.7790	0.8094

Table 1. Precision, Recall, F1 and Accuracy of the naive bayes classifiers on the development set, when trained on each feature set.

As we can see, the second feature set performed the best overall, achieving the highest F1 score, representing the balance between the model's precision and recall, of 0.8021, and an accuracy of approximately 82.5%. I then evaluated feature set 2 on the test set. Again, sklearn's multinomial naïve bayes classifier achieved the same scores.

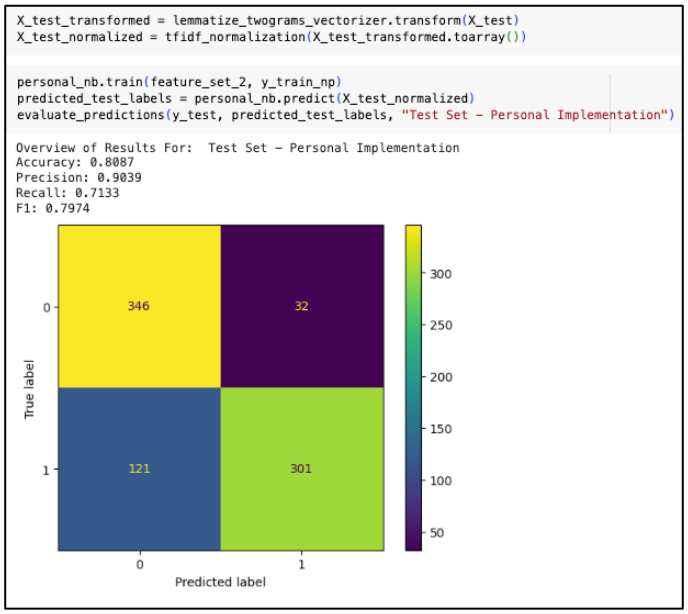


Figure 16. Performance of personal implementation of Naïve Bayes of test set on feature set 2.

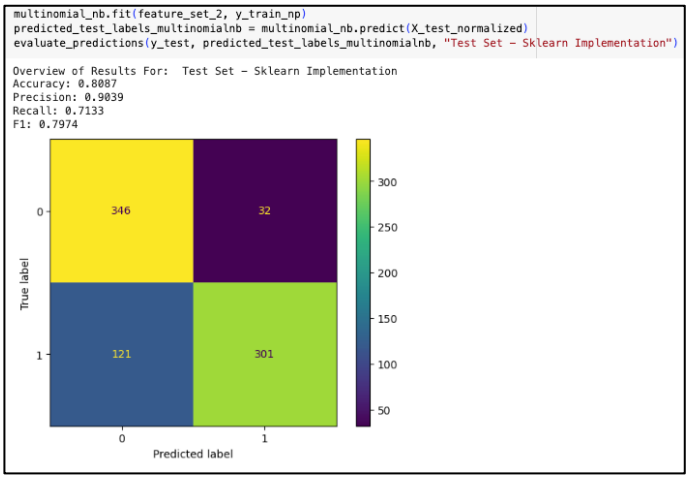


Figure 17. Performance of sklearn's multinomial naïve bayes on feature set 2.

3.7 Performance of Logistic Regression Classifier on the 3 Feature Sets

The three feature sets were then evaluated using sklearn's logistic regression model. This was done using the development set.

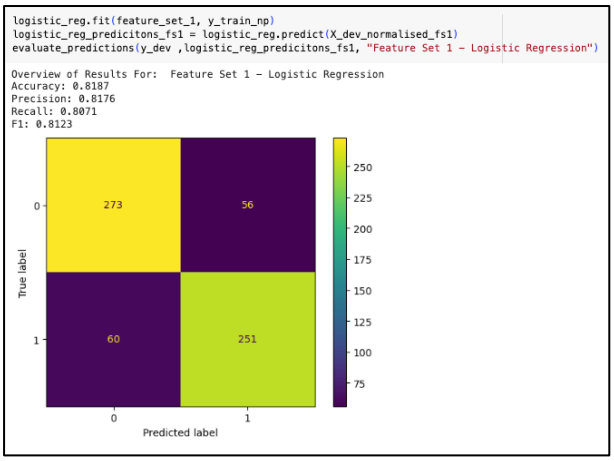


Figure 18. Performance of sklearn's logistic regression model on feature set 1, which contained only unigrams.

Results of Logistic Regression Classifier					
Feature Set	N-Grams Contained in Set	Precision	Recall	F1	Accuracy
1	Unigrams	0.8176	0.8071	0.8123	0.8187
2	Unigrams, Bigrams	0.8505	0.7685	0.8074	0.8219
3	Unigrams, Bigrams, Trigrams	0.8858	0.7235	0.7965	0.8203

Table 2. Precision, Recall, F1 and Accuracy of sklearn's logistic regression classifier on the development set, when trained on each feature set.

Again, the second feature set 2 performed the best overall based on accuracy, although if you were more concerned with the model's recall performance you would conclude that the model trained on the first feature set in the most useful. Nevertheless, feature set 2 was selected to for then training a logistic regression model after tuning hyperparameters. The logistic regression classifier has a few important hyperparameters to try changing. These include the C value, which controls the regularization strength of the model, the solver, specifying the algorithm used to best optimise the model, and the penalty, specifying the type of regularization applied, as seen in sklearn's documentation (scikit-learn, n.d).The default values set C to 1, the solver to 'lbfgs' and the penalty to 'l2'. I performed 5 different hyperparameters changes, with an example in figure 19, and the overall results are presented in table 3.

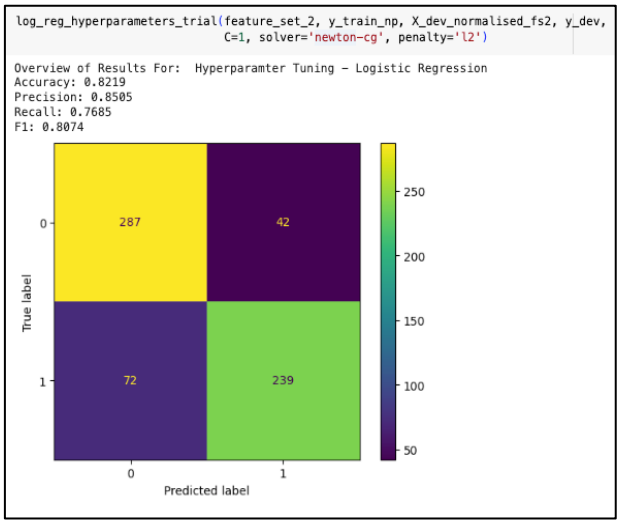


Figure 19. Performance of sklearn's logistic regression model on feature set 2 with 'newton-cg' as its solver.

As seen, where possible I only changed one hyperparameter at a time so that any changes in model performance could be attributed to just this change. For reference, the code that was used in evaluating different hyperparameters is shown below.

```
def log_reg_hyperparameters_trial(X_train, Y_train, X_dev, Y_dev, C=1, solver='lbfgs', penalty='l2'):
  log_reg = LogisticRegression(C=C, solver=solver, penalty=penalty)
  log_reg.fit(X_train, Y_train)
  predictions = log_reg.predict(X_dev)
  evaluate_predictions(Y_dev ,predictions, "Hyperparamter Tuning - Logistic Regression")
```

Figure 20. Helper function to test out different hyperparameters for logistic regression.

Results of Logistic Regression Classifier on Feature Set 2 with Different Hyperparameters							
Combination	C	Solver	Penalty	Precision	Recall	F1	Accuracy
Default	1	lbfgs	l2	0.8505	0.7685	0.8074	0.8219
1	1	newton-cg	l2	0.8505	0.7685	0.8074	0.8219
2	1	sag	l2	0.8505	0.7685	0.8074	0.8219
3	1	saga	l2	0.8505	0.7685	0.8074	0.8219
4	1	liblinear	l1	0.5765	0.8842	0.6980	0.6281
5	100	lbfgs	l2	0.8388	0.8199	0.8293	0.8359

Table 3. Precision, Recall, F1 and Accuracy of sklearn's logistic regression classifier with a variety of different hyperparameters, evaluated on the development set

Lastly, the best performing model, which is combination 5 in the table, was evaluated using the test set. We can see that the model performs well, achieving an accuracy of 86.5% and an F1 score of 0.87.

```
log_reg = LogisticRegression(C=100, solver='lbfgs', penalty='l2')
log_reg.fit(feature_set_2, y_train_np)
predicted_test_labels_log_reg = log_reg.predict(X_test_normalized)
evaluate_predictions(y_test, predicted_test_labels_log_reg , "Test Set - Logistic Regression")

Overview of Results For:  Test Set - Logistic Regression
Accuracy: 0.8650
Precision: 0.8829
Recall: 0.8578
F1: 0.8702
```

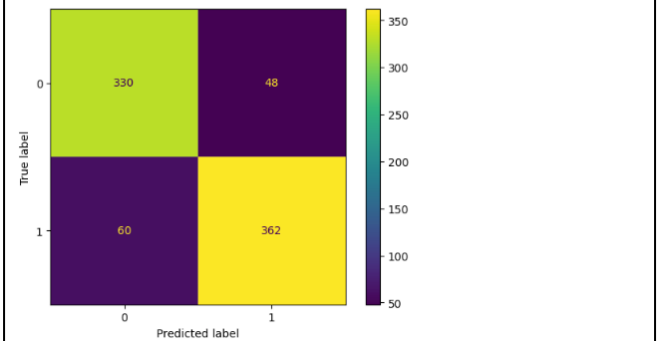


Figure 21. Performance of a logistic regression classifier with hyperparameter combination 5 (from table 3) on the test set, after being trained on feature set 2.

3.8 Performance of SVM Classifier on the 3 Feature Sets

Similar experiments were then carried out on sklearn's SVM classifier. First, as before, the SVM classifier was evaluated on the development set after being trained on each feature set. As seen from Table 4, feature 1 performed the best in this case, achieving the same accuracy as the model trained using feature set 2 achieved, but with a higher F1 score. The code to test different hyperparameters is shown in figure 23. In tuning hyperparameters, then, feature set 1 was used in for evaluation. Similar to logistic regression classifiers, SVMs also have a C hyperparameter to change. Unlike logistic regression, we have a kernel parameter, that is used by the model to transform our training data into a higher dimension.

```
from sklearn import svm
svm_model = svm.SVC()
svm_model.fit(feature_set_1, y_train_np)
svm_model_predicitons_fs1 = svm_model.predict(X_dev_normalised_fs1)
evaluate_predictions(y_dev ,svm_model_predicitons_fs1, "Feature Set 1 - SVM")

Overview of Results For:  Feature Set 1 - SVM
Accuracy: 0.8125
Precision: 0.7903
Recall: 0.8360
F1: 0.8125
```

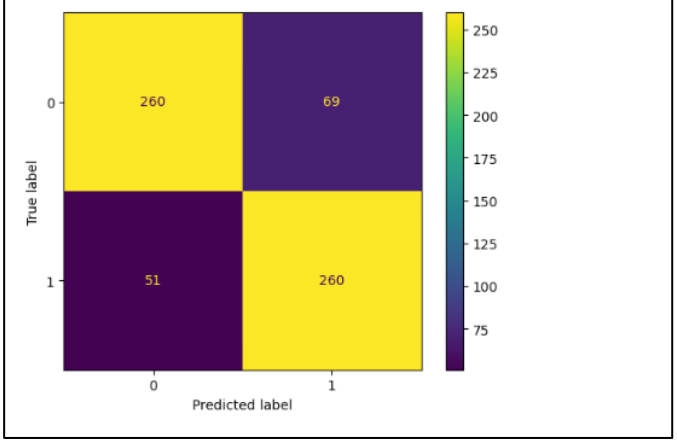


Figure 22. Performance of the SVM classifier on feature set 1.

Results of SVM Classifier					
Feature Set	N-Grams Contained in Set	Precision	Recall	F1	Accuracy
1	Unigrams	0.7903	0.8360	0.8125	0.8125
2	Unigrams, Bigrams	0.8131	0.7974	0.8052	0.8125
3	Unigrams, Bigrams, Trigrams	0.8223	0.7588	0.7893	0.8031

Table 4. Performance of the SVM classifier on all three feature sets.

```
[ ] def svm_hyperparameters_trial(X_train, Y_train, X_dev, Y_dev, C=1, kernel='rbf', degree=3, gamma='scale'):
      svm_model = svm.SVC(C=C, kernel=kernel, degree=degree, gamma=gamma)
      svm_model.fit(X_train, Y_train)
      predictions = svm_model.predict(X_dev)
      evaluate_predictions(Y_dev , predictions, "Hyperparamter Tuning - SVMs")
```

Figure 23. Helper function to test out different hyperparameters for the SVM model.

We also have a degree hyperparameter, that controls the degree of the polynomial that defines the decision boundary between data points, and a gamma hyperparameter which also effects how the decision boundary is created. Such parameters are noted and explained further in sklearn's documentation (scikit-learn, n.d). The performance of each SVM classifier, with different hyperparameters is shown in table 5.

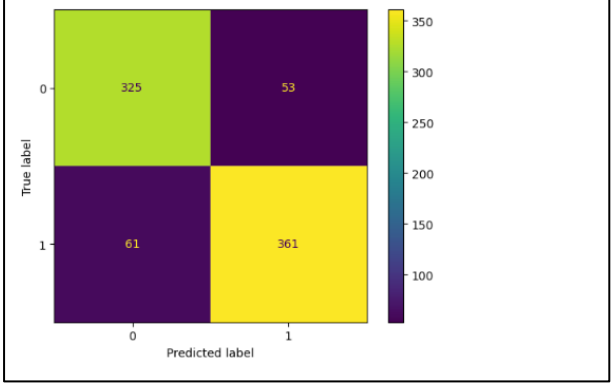
Results of SVM Classifier on Feature Set 1 with Different Hyperparameters								
Combination	C	Kernel	Degree	Gamma	Precision	Recall	F1	Accuracy
Default	1	rbf	3	scale	0.7903	0.8360	0.8125	0.8125
1	1	sigmoid	3	scale	0.8119	0.8328	0.8222	0.8250
2	100	sigmoid	3	scale	0.7342	0.7460	0.7227	0.7375
3	1	poly	3	scale	0.8000	0.0129	0.0253	0.5188
4	1	sigmoid	5	scale	0.8119	0.8328	0.8222	0.8250
5	0.1	sigmoid	3	auto	0.9388	0.2958	0.4499	0.6484

Table 5. Precision, Recall, F1 and Accuracy of sklearn's SVM classifier with a variety of different hyperparameters, evaluated on the development set

As seen, combination 1 and 4 yielded the best results, both using the sigmoid kernel. Both could be used for the test set, but I have evaluated the test using combination 1, given that a higher degree polynomial is more likely to result in the decision boundary overfitting the training data. The evaluation of an SVM model with hyperparameters specified in combination 1 on the test set can be seen below. The model performs well, achieving an accuracy of 86% and an F1 score of 0.86.

```
svm_for_test_set = svm.SVC(C=1, kernel='sigmoid', degree=3, gamma='scale')
svm_for_test_set.fit(feature_set_1, y_train_np)

X_test_transformed_svm = lemmatize_onegrams_vectorizer.transform(X_test)
X_test_normalized_svm = tfidf_normalization(X_test_transformed_svm.toarray())

predicted_test_labels_svm_test = svm_for_test_set.predict(X_test_normalized_svm)
evaluate_predictions(y_test, predicted_test_labels_svm_test , "Test Set - SVM")
```

```
Overview of Results For:  Test Set - SVM
Accuracy: 0.8575
Precision: 0.8720
Recall: 0.8555
F1: 0.8636
```



Results of all Classifiers on the Test Set						
Model	Feature Set (N-Grams Considered)	Hyperparameter Combination	Precision	Recall	F1	Accuracy
Naïve Bayes (Personal)	2 (Unigrams, Bigrams)	N/A	0.9039	0.7133	0.7974	0.8087
Naïve Bayes (Sklearn)	2 (Unigrams, Bigrams)	N/A	0.9039	0.7133	0.7974	0.8087
Logistic Regression	2 (Unigrams, Bigrams)	5	0.8829	0.8578	0.8702	0.8650
Support Vector Machine	1 (Unigrams)	1	0.8720	0.8555	0.8636	0.8575

Table 6. Precision, Recall, F1 and Accuracy of all considered classifiers on the test set, using the feature set and hyperparameter combination that resulted in the best development set performance of each model.

4 Discussion

In this assignment, I created three different feature sets to test the performance of different classification models on predicting whether a movie review is positive or negative. The features were all generated through being tokenized, having punctuation and stop words removed, and then the words were lemmatized. Three sets were then generated, with feature set 1 only containing unigrams, set 2 containing unigrams and bigrams, and lastly set 3 containing unigrams, bigrams, and trigrams. As we can see, in all classification models except the SVM, feature set 2 yielded the best performance. This is perhaps because the use of bigrams in addition to unigrams could better capture phrases and context within the review, while trigrams did not catch any useful information.

Regarding the Naïve Bayes classifier, the best accuracy obtained was approximately 81%. This is the lowest of all the classifiers and may be as a result of the assumption of independence between words. In sentiment analysis this assumption may be especially inaccurate As one example, negating a word through using 'not' may completely change the sentiment, though not always. Additionally, we have to note that I did not experiment with hyperparameter tuning with my implementation of Naïve Bayes or sklearn's implementation. The only parameter to tune, for my implementation, would have been the alpha value, but this still may have affected the performance.

The SVM model performed very well too, with an accuracy and F1 score of approximately 86% of the test set. Interestingly, this was done through using feature set 1, which only contained unigrams. This may be because the data was more separable when only words were considered. Since the data set is also a lot smaller, finding a decision boundary between points may also have been easier in this case, without risking overfitting the data. Moreover, the SVM model also underwent hyperparameter tuning, whereby the sigmoid kernel, making use of the sigmoid function that the logistic regression also uses, was chosen as the best hyperparameter setting for this task.

The logistic regression classifier, being trained on feature set 2, achieved the highest accuracy and F1 score of approximately 87%. Interestingly, this was when the C hyperparameter was set to 100. A higher C value here is representing low regularization, meaning the model fits the training data more closely, placing a greater focus on achieving a high accuracy when training. Though this risks overfitting, it appears that the training data accurately represents features typical of a positive or negative review since the model still performs well on the test set. As such, the logistic regression classifier may have performed best here since it was most suited at capturing patterns in the training set, through low regularization, that are common of negative and positive reviews.

Lastly, we can note that the use of more than just words (unigrams) as features is clearly effective, given that both the Naïve Bayes and Logistic Regression model performed better on feature set 2. The computational cost, on this small scale, seems worth it.

5 Conclusions and Future Work

Overall, we found that considering ranges of n-grams, for sentiment analysis of movie reviews, was an effective method for improving performance for a Naïve Bayes classifier and a logistic regression classifier, while a SVM worked best in this context on unigrams only. With the best performing model being the logistic regression classifier, with low regularization, it seems that the training data provided an accurate reflection of

the characteristics of positive and negative movie reviews.

Given the scope of the problem, there is a lot left to explore. To better perform hyperparameter optimisation, one could make use of sklearn's GridSearchCV functionality, which can perform an exhausting comparison between different hyperparameters on a model. I have concluded with a logistic regression model that performed the best, but there may be a set of hyperparameters for the SVM model that results in even better results. One could also explore the use of different features, other than just words. For example, I removed punctuation from all feature sets, but the use of a punctuation mark, for example, may be a good indicator of a strong positive or negative sentiment depending on words it is used around too. Also, deep learning models have also not been discussed in this assignment, and so in the future making use of such models, such as BERT, to see if performance is dramatically changed is also interesting.

References

Ahuja, R., Chug, A., Kohli, S., Gupta, S. and Ahuja, P. (2019). The Impact of Features Extraction on the Sentiment Analysis. *Procedia Computer Science*, 152, pp.341–348. doi:https://doi.org/10.1016/j.procs.2019.05.008.

Baid, P., Gupta, A. and Chaplot, N. (2017). Sentiment Analysis of Movie Reviews using Machine Learning Techniques. *International Journal of Computer Applications*, [online] 179(7), pp.45–49. doi:https://doi.org/10.5120/ijca2017916005.

Birjali, M., Kasri, M. and Beni-Hssane, A. (2021). A comprehensive survey on sentiment analysis: Approaches, challenges and trends. *Knowledge-Based Systems*, 226(1), p.107134. doi:https://doi.org/10.1016/j.knosys.2021.107134.

Dey, L., Chakraborty, S., Biswas, A., Bose, B. and Tiwari, S. (2016). Sentiment Analysis of Review Datasets Using Naïve Bayes' and K-NN Classifier. *International Journal of Information Engineering and Electronic Business*, [online] 8(4), pp.54–62. doi:https://doi.org/10.5815/ijieeb.2016.04.07.

Feldman, R. (2013). Techniques and applications for sentiment analysis. *Communications of the ACM*, 56(4), p.82. doi:https://doi.org/10.1145/2436256.2436274.

Li, J., Guo, J., Zhu, Y., Sheng, X., Jiang, D., Ren, B. and Xu, L. (2022). Sequence-to-Action: Grammatical Error Correction with Action Guided Sequence Generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, [online] 36(10), pp.10974–10982. doi:https://doi.org/10.1609/aaai.v36i10.21345.

Medhat, W., Hassan, A. and Korashy, H. (2014). Sentiment analysis algorithms and applications: A survey. *Ain Shams Engineering Journal*, 5(4), pp.1093–1113.

Moraes, R., Valiati, J.F. and Gavião Neto, W.P. (2013). Document-level sentiment classification: An empirical comparison between SVM and ANN. *Expert Systems with Applications*, 40(2), pp.621–633. doi:https://doi.org/10.1016/j.eswa.2012.07.059.

P. Tanawongsuwan (2010). Product review sentiment classification using parts of speech. In: *2010 3rd International Conference on Computer Science and Information Technology*. pp.424–427. doi:https://doi.org/10.1109/ICCSIT.2010.5563883.

Psomakelis, E., Tserpes, K., Anagnostopoulos, D. and Varvarigou, T. (2015). Comparing methods for Twitter Sentiment Analysis. *arXiv:1505.02973 [cs]*. [online] Available at: https://arxiv.org/abs/1505.02973.

scikit-learn (n.d.). *sklearn.feature_extraction.text.CountVectorizer — scikit-learn 0.20.3 Documentation*. [online] Scikit-learn.org. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html [Accessed 10 Dec. 2023].

scikit-learn (n.d.). *sklearn.linear_model.LogisticRegression — scikit-learn 0.21.2 documentation*. [online] Scikit-learn.org. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html [Accessed 10 Dec. 2023].

scikit-learn (n.d.). *sklearn.model_selection.train_test_split — scikit-learn 0.20.3 documentation*. [online] Scikit-learn.org. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html [Accessed 7 Dec. 2023].

scikit-learn (n.d.). *sklearn.svm.SVC — scikit-learn 0.22 documentation*. [online] Scikit-learn.org. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html [Accessed 10 Dec. 2023].

Shahnawaz and Astya, P. (2017). *Sentiment analysis: Approaches and open issues*. [online] IEEE Xplore. doi:https://doi.org/10.1109/CCAA.2017.8229791.

Wankhade, M., Rao, A.C.S. and Kulkarni, C. (2022). A survey on sentiment analysis methods, applications, and challenges. *Artificial Intelligence Review*, 55(55). doi:https://doi.org/10.1007/s10462-022-10144-1.

Yadav, A. and Vishwakarma, D.K. (2019). Sentiment analysis using deep learning architectures: a review. *Artificial Intelligence Review*, 53(6). doi:https://doi.org/10.1007/s10462-019-09794-5.