Sentiment Analysis

Abstract

This project focuses on developing a Sentiment Analysis application, targeting automated identification of opinion polarity within text, specifically in movie reviews. The project uses a subset of 2,000 positive and 2,000 negative movie reviews from the Large Movie Review Dataset, sourced from the Internet Movie Database (IMDb) (Maas, 2011). Reviews are categorized based on star ratings, with scores <=4 representing negative and scores >=7 representing positive. I will investigate the Sentiment Analysis pipeline, covering text classification, feature extraction, and model selection and evaluation. This project explores various Sentiment Analysis models and techniques, evaluating the performance of each. The goal of this project is to create a robust system capable of accurately detecting sentiment polarity. By implementing natural language processing (NLP) techniques, machine learning models, and leveraging the labelled dataset, the developed application aims to provide a practical solution for analysing sentiment polarity in diverse textual reviews. This project addresses the growing demand for automated sentiment analysis tools, particularly for movie reviews, offering a valuable tool for researchers, businesses, and enthusiasts in the field.

1 Introduction and Motivation

Sentiment Analysis is a type of text classification and is a vital area of natural language processing (NLP), identifying the overall sentiment of a piece of text. As technology continues to evolve, the growth of automated sentiment analysis becomes more important for gaining insights into public opinions, customer feedback and user sentiments (Gupta, 2018). Sentiment analysis is often presented in a binary context, distinguishing between positive and negative sentiments. However, it can also take on various formats, including multi-class classifications, regression models, or ranking systems. This project aims to explore the development of a sentiment analysis application specifically targeting the analysis of movie reviews from IMDb.

Sentiment analysis uses machine learning models to perform text analysis of human language, which will detect whether the overall sentiment of a document is positive, negative or neutral. It generally follows these steps which will be further discussed in the Experiments and Results section:

1. Data collection
2. Preprocess text
3. Feature extraction
4. Model selection
5. Train model (if using ML algorithm)
6. Sentiment classification
7. Model evaluation

1.1 Importance of Sentiment Analysis

Understanding sentiment in textual reviews provides valuable insights not only for businesses but also for researchers. It allows them to gain insight into how customers perceive and experience their products and brands. Businesses may not have the time or resources to look into and analyse every single piece of data relating to their product. Therefore, automated sentiment analysis algorithms are used widely to provide real-time feedback. This is then used to improve products or services, thus improving customer experience and satisfaction levels.

Specifically, in the context of movie reviews, sentiment analysis can allow filmmakers and businesses to understand audience reactions, or produce personalised content recommendations. Consequently, it contributes to informed decision-making in the dynamic entertainment industry (Barney, 2023).

1.2 Challenges/Difficulties in Sentiment Analysis

While sentiment analysis holds significance, it is important to acknowledge the difficulties it faces. Context-dependent errors such as sarcasm can be confusing as sarcasm uses positive words to express negative sentiment, for example, "*Wow, that movie was so good, I totally didn't fall asleep*". In these cases, sentiment analysis tools will wrongly classify the review as positive, rather than negative. Similarly reviews with a neutral sentiment pose a polarity problem for systems and are often misidentified as the system may classify the neutral comment as a negative sentiment. For example, if a customer received the wrong colour product and submitted a review, "*The product was* *blue*," it could be classified as neutral when in fact it should be negative. Furthermore, there is the issue of polysemy, where a word has more than one meaning. Without context, it makes it challenging for the algorithm to figure out the intended meaning, therefore, it may misclassify the sentiment. For example, "*head*" has 2 different meanings in the following sentences; the head of the sales team vs. wearing an earbud hurts the head. Named-entity recognition poses a difficulty as well, it is when the algorithm is unable to recognise the meaning of a word in its context. For example, the use of the word "*Lincoln*" may refer to the former United States President, the film or a penny.

Negation detection is also one of the biggest challenges in sentiment analysis. Algorithms tend to classify sentences containing negations, for example, *"no, not, -non, -less, -dis"* as negative sentiments when sometimes the sentence has phrased them to be positive.

Lastly, multilingual data is common in product reviews worldwide. Customers writing reviews in multiple languages can confuse sentiment analysis tools as they are mainly trained to categorise words in a specific language, usually English. Therefore, when the review is translated into one language, some sentiments may get lost in the translation process. This is one of the biggest challenges when performing sentiment analysis on non-English reviews.

In this project, movie reviews contain subjective expressions, therefore, it can be seen as an ideal but also a challenging aspect for sentiment analysis. Overcoming these challenges requires advanced NLP techniques and robust machine-learning models (Yılmaz, 2023).

1.3 Diverse/Different Methods Available

There are various existing methods for sentiment analysis, ranging from traditional machine learning approaches to state-of-the-art deep learning models. Rule-based systems and support vector machines (SVMs) are examples of these approaches. This project explores machine learning approaches, investigating their effectiveness in accurately categorising movie reviews. Existing methods that perform sentiment analysis are further detailed in the Related Works section.

1.4 Testing Objectives

The primary objective of this project is to develop a robust sentiment analysis system that accurately classifies the polarity of movie reviews as positive or negative. I aim to evaluate the performance of different models, namely Naive Bayes, SGD-based classification and SVMs using the test data set I will create in my data splits. The models will be evaluated on three different feature sets to identify the most effective approach.

1.5 Exploring Novel Approaches

From the second week of labs, I explored term-weighting techniques which taught me `string.punctuation` is used to remove punctuations from a list of words. However, when pre-processing my data, I ended up trying to use regular expressions instead to remove punctuation from the text. This is further explained in the Feature Selection part of the Experiments Section.

1.6 Methodology and Evaluation

The methodology involves implementing NLP techniques and machine learning models on the movie reviews dataset. The project will test and evaluate the performance of each method – Naive Bayes, SGD-based classification, SVMs – against established metrics – the three feature sets I will extract. Through this, I aim to gain insights into the strengths and weaknesses of each approach. Thus, providing insights into their efficiency in handling the complexities of sentiment analysis. The methodology and evaluation are explored in more depth in the Experiments and Results section.

2 Related work

2.1 Lexicon-based Methods

Lexicon-based methods are pre-developed manually and refer to analysing semantic and syntactic patterns. They often involve the use of pre-developed dictionaries (e.g., WorldNet) or lexicons that associate words with sentiment orientation (positive, negative, neutral), making them easily accessible. To analyse semantic patterns, a dictionary is generated by tagging words and assigning sentiment scores or labels to words based on the corresponding semantic meaning. While syntactic patterns involve the consideration of syntactic patterns. Training data is not commonly used in lexicon-based methods, especially if a dictionary-based approach is used, as the tags are determined manually. However, Lexicon-based sentiment analysis methods usually do not identify sarcasm, negation, grammar mistakes, misspellings, or irony. Additionally, a word is labelled as the same no matter the context (Kannan et al., 2016).

2.2 Deep Learning Methods

Due to deep learning methods being able to learn representation of textual data, it has become a popular method for sentiment analysis. Textual data is preprocessed and then encoded using pre-trained embeddings such as GloVe and Word2Vec. These embeddings are then fed into deep learning models such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), long short-term memory (LSTM), and gated recurrent units (GRUs) for representation learning and classification (Tan et al., 2023).

2.3 Rule-Based Methods

The rule-based approach uses predefined rules to identify sentiment patterns, looking for opinion words in a text and then classifying them based on the number of positive and negative words. It considers different grammatical structure rules for classification such as dictionary polarity, negation words, booster words, idioms, emoticons, mixed opinions etc. Rule-based approaches offer interpretability and are often tailored for specific applications or domains (Collomb et al., 2014).

2.4 Hybrid Methods

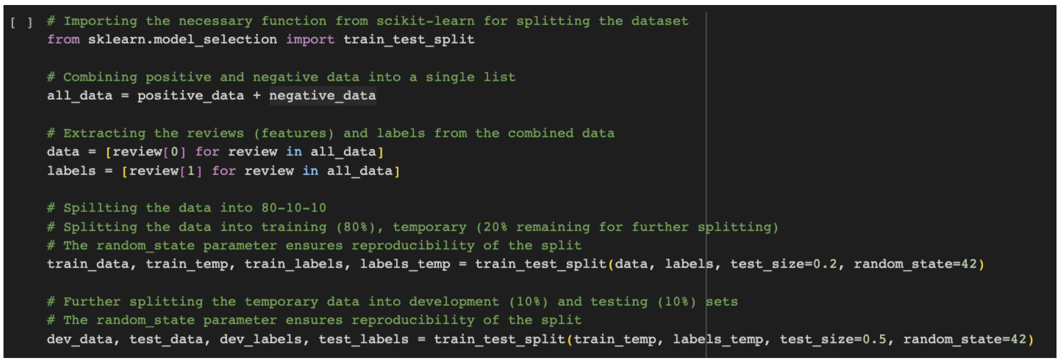
Hybrid methods combine the advantages of different methods to enhance sentiment analysis accuracy, compensating for the other approaches' flaws. Integrating lexicon-based methods with machine learning or rule-based systems can leverage the strengths of each approach, thus, providing a more robust sentiment classification. The integration of methods can occur in parallel or at different stages of the sentiment analysis pipeline (Yılmaz, 2023).

3 Experiments and Results

3.1 Data Splits

Once the movie reviews dataset has been loaded into the system, all the positive and negative data is then combined into one list (`all_data`) to give it a uniform representation. Once all the data is in one list, I extract the reviews and labels from the combined data list. The data list (`data`) contains all the reviews, and the labels list (`labels`) contains the corresponding labels (positive or negative). I then split the data into segments — training, development, and testing — employing an 80-10-10 data split. I chose this particular data split as it allows for a sufficiently large dataset for training, while also setting aside substantial portions for evaluating and validating the model's performance. I split the data by using the `train_test_split` function imported by the `sklearn.model_selection` library.

- `train_data, dev_data, test_data`: reviews for each data split
- `train_labels, dev_labels, test_labels`: respective labels



```
[ ] # Importing the necessary function from scikit-learn for splitting the dataset
    from sklearn.model_selection import train_test_split

    # Combining positive and negative data into a single list
    all_data = positive_data + negative_data

    # Extracting the reviews (features) and labels from the combined data
    data = [review[0] for review in all_data]
    labels = [review[1] for review in all_data]

    # Spillting the data into 80-10-10
    # Splitting the data into training (80%), temporary (20% remaining for further splitting)
    # The random_state parameter ensures reproducibility of the split
    train_data, train_temp, train_labels, labels_temp = train_test_split(data, labels, test_size=0.2, random_state=42)

    # Further splitting the temporary data into development (10%) and testing (10%) sets
    # The random_state parameter ensures reproducibility of the split
    dev_data, test_data, dev_labels, test_labels = train_test_split(train_temp, labels_temp, test_size=0.5, random_state=42)
```

Figure 1: Code for splitting data

3.2 Feature selection

I created three different sets of features, each using a combination of the following:

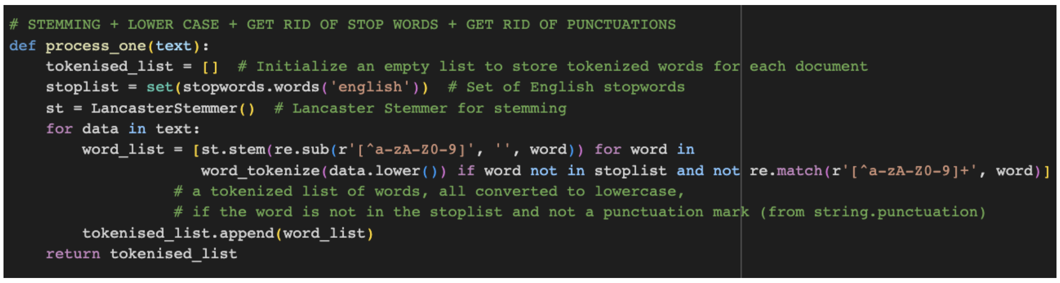
- Stemming: returns the common stem (root form)
  - Used when the linguistic information is less relevant
  - `LancasterStemmer` imported from `nltk.stem`
- Lemmatising: returns the lemma (root meaning)
  - Used when the linguistic information is relevant
  - `WordNetLemmatizer` imported from `nltk.stem`

- Removing stop words: common words that hold less information content
    - `stopwords` imported from `nltk.corpus`
- Removing punctuations
    - `string.punctuation` resulted in low accuracy scores, therefore, I used regular expressions imported from `re`
    - `re.sub(r'[^a-zA-Z0-9]', '', word)`: replace non-alphanumeric character with an empty string
    - `re.match(r'[^a-zA-Z0-9]+', word)`: filters out words that start with non-alphanumeric characters
- Lowercase conversion
    - `lower()`

This diversity in feature sets allows me to explore different linguistic characteristics in the text data. For all feature sets, I tokenise the text to extract the particular features, converting the raw text into a structured format, using `word_tokenize`. I created three separate functions for the different feature selections.

3.2.1 First Feature Set

The first set of features is generated through a combination of stemming, lowercase conversion, removal of stop words, and removal of punctuation marks. This method aims to get rid of any common language; words that typically carry little semantic meaning. More importantly, it captures the root forms of the words.

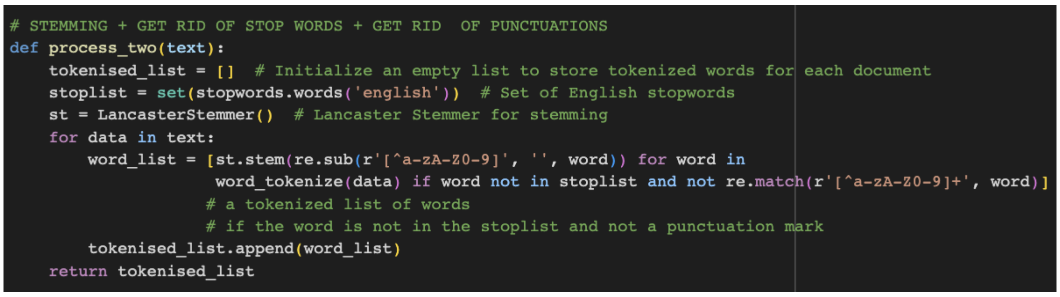


```python
# STEMMING + LOWER CASE + GET RID OF STOP WORDS + GET RID OF PUNCTUATIONS
def process_one(text):
    tokenised_list = []  # Initialize an empty list to store tokenized words for each document
    stoplist = set(stopwords.words('english'))  # Set of English stopwords
    st = LancasterStemmer()  # Lancaster Stemmer for stemming
    for data in text:
        word_list = [st.stem(re.sub(r'[^a-zA-Z0-9]', '', word)) for word in
                     word_tokenize(data.lower()) if word not in stoplist and not re.match(r'[^a-zA-Z0-9]+', word)]
                   # a tokenized list of words, all converted to lowercase,
                   # if the word is not in the stoplist and not a punctuation mark (from string.punctuation)
        tokenised_list.append(word_list)
    return tokenised_list
```

Figure 2: Extracting the first feature set

3.2.2 Second Feature Set

The second set of features focuses on stemming and exclusion of stopwords and punctuation marks. This method keeps the fundamental structure of the text, simplifying it by stemming and eliminating stopwords, with the exception of punctuation marks.

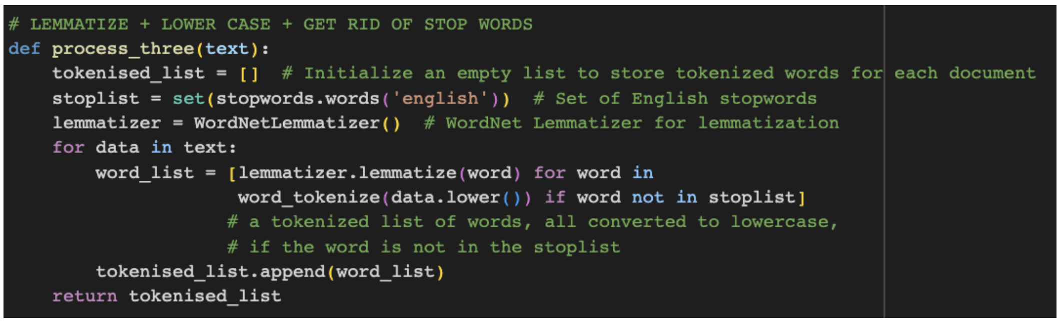


```python
# STEMMING + GET RID OF STOP WORDS + GET RID  OF PUNCTUATIONS
def process_two(text):
    tokenised_list = []  # Initialize an empty list to store tokenized words for each document
    stoplist = set(stopwords.words('english'))  # Set of English stopwords
    st = LancasterStemmer()  # Lancaster Stemmer for stemming
    for data in text:
        word_list = [st.stem(re.sub(r'[^a-zA-Z0-9]', '', word)) for word in
                     word_tokenize(data) if word not in stoplist and not re.match(r'[^a-zA-Z0-9]+', word)]
                    # a tokenized list of words
                    # if the word is not in the stoplist and not a punctuation mark
        tokenised_list.append(word_list)
    return tokenised_list
```

Figure 3: Extracting the second feature set

3.2.3 Third Feature Set

Lastly, the third set of features involves lemmatisation, lowercase conversion, and excluding stop words. The goal of this method is to create a detailed representation of the text, emphasising the essential meaning of words and improving the model's ability to understand and generalize across different contexts.



```python
# LEMMATIZE + LOWER CASE + GET RID OF STOP WORDS
def process_three(text):
    tokenised_list = []  # Initialize an empty list to store tokenized words for each document
    stoplist = set(stopwords.words('english'))  # Set of English stopwords
    lemmatizer = WordNetLemmatizer()  # WordNet Lemmatizer for lemmatization
    for data in text:
        word_list = [lemmatizer.lemmatize(word) for word in
                     word_tokenize(data.lower()) if word not in stoplist]
                    # a tokenized list of words, all converted to lowercase,
                    # if the word is not in the stoplist
        tokenised_list.append(word_list)
    return tokenised_list
```

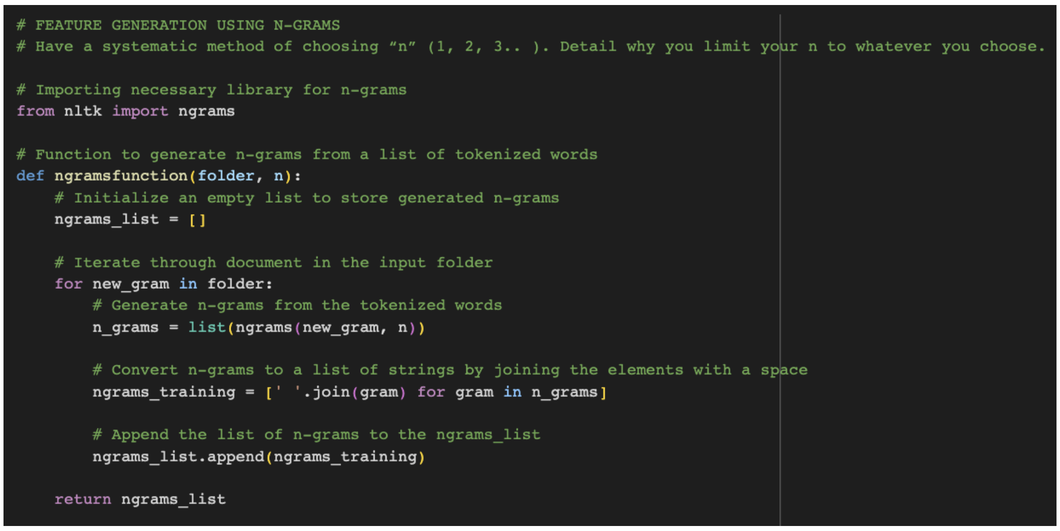
Figure 4: Extracting the third feature set

The input parameter of each function is the list of reviews. The function outputs a list of tokenised and processed words.

3.3 Feature Generation using n-grams

To generate my features using n-grams, I implemented a systematic method for choosing the value of "n", considering the range of 1, 2, and 3. Again, I used the `nltk` library and imported the `ngrams` function. To produce the n-grams, each document, i.e., review, in the folder is iterated over and then converted into a list of strings. There are two input parameters, `folder`, and `n`

- `folder`: the list of tokenised and processed words from the feature selection process
- `n`: the number of consecutive words grouped as a single feature



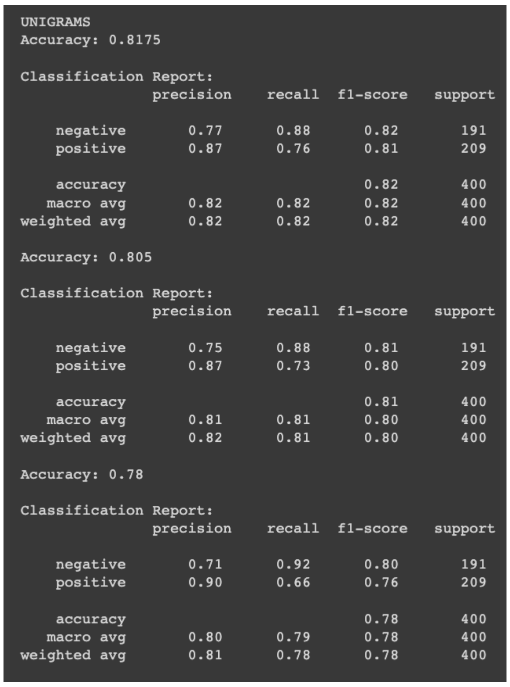
```
# FEATURE GENERATION USING N-GRAMS
# Have a systematic method of choosing "n" (1, 2, 3.. ). Detail why you limit your n to whatever you choose.

# Importing necessary library for n-grams
from nltk import ngrams

# Function to generate n-grams from a list of tokenized words
def ngramsfunction(folder, n):
    # Initialize an empty list to store generated n-grams
    ngrams_list = []

    # Iterate through document in the input folder
    for new_gram in folder:
        # Generate n-grams from the tokenized words
        n_grams = list(ngrams(new_gram, n))

        # Convert n-grams to a list of strings by joining the elements with a space
        ngrams_training = [' '.join(gram) for gram in n_grams]

        # Append the list of n-grams to the ngrams_list
        ngrams_list.append(ngrams_training)

    return ngrams_list
```

Figure 5: Generating n-grams

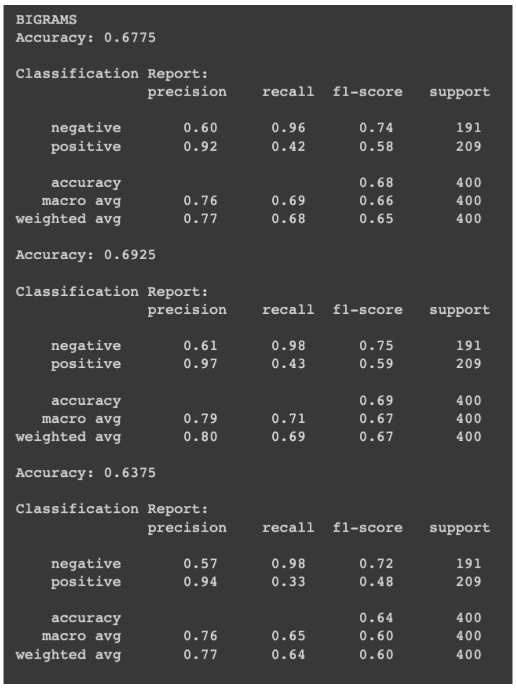
The systematic exploration of different "n" values allows for the creation of diverse n-gram features, capturing varying degrees of linguistic context within the text. Smaller values of "n" (e.g., unigrams or bigrams) help keep things manageable for the model, so it doesn't get too confused with too many features and potentially make mistakes. On the other hand, using larger values (e.g., trigrams or higher) helps the model understand more complex patterns and connections between words in the text.

After testing the different n-grams on the in-built Naïve Bayes classifier, I chose to focus on unigrams (1-gram) as it gave me the best results. As shown in Figure 6, the higher the value of "n", the lower the accuracy rate. Resulting in trigrams giving the worst accuracy results. The change in accuracy rates is reflected in all 3 feature sets.

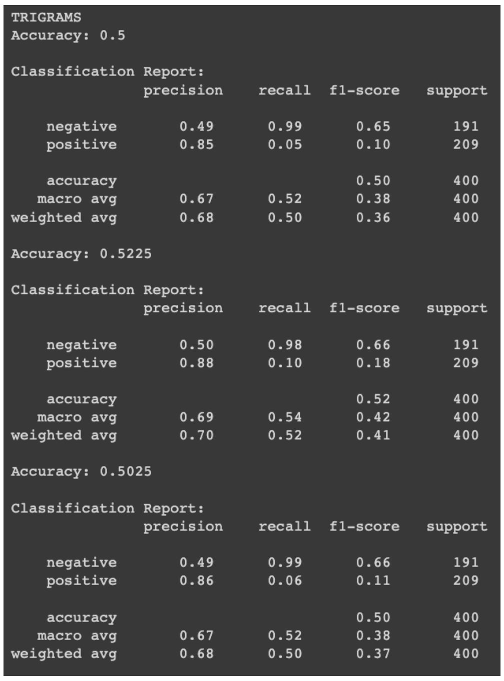
The choice of using unigrams allows for the examination of individual words in isolation, providing insight into the basic building blocks of the language.



```
UNIGRAMS
Accuracy: 0.8175

Classification Report:
               precision    recall  f1-score   support

    negative       0.77      0.88      0.82       191
    positive       0.87      0.76      0.81       209

    accuracy                           0.82       400
   macro avg       0.82      0.82      0.82       400
weighted avg       0.82      0.82      0.82       400

Accuracy: 0.805

Classification Report:
               precision    recall  f1-score   support

    negative       0.75      0.88      0.81       191
    positive       0.87      0.73      0.80       209

    accuracy                           0.81       400
   macro avg       0.81      0.81      0.80       400
weighted avg       0.82      0.81      0.80       400

Accuracy: 0.78

Classification Report:
               precision    recall  f1-score   support

    negative       0.71      0.92      0.80       191
    positive       0.90      0.66      0.76       209

    accuracy                           0.78       400
   macro avg       0.80      0.79      0.78       400
weighted avg       0.81      0.78      0.78       400
```



```
BIGRAMS
Accuracy: 0.6775

Classification Report:
               precision    recall  f1-score   support

    negative       0.60      0.96      0.74       191
    positive       0.92      0.42      0.58       209

    accuracy                           0.68       400
   macro avg       0.76      0.69      0.66       400
weighted avg       0.77      0.68      0.65       400

Accuracy: 0.6925

Classification Report:
               precision    recall  f1-score   support

    negative       0.61      0.98      0.75       191
    positive       0.97      0.43      0.59       209

    accuracy                           0.69       400
   macro avg       0.79      0.71      0.67       400
weighted avg       0.80      0.69      0.67       400

Accuracy: 0.6375

Classification Report:
               precision    recall  f1-score   support

    negative       0.57      0.98      0.72       191
    positive       0.94      0.33      0.48       209

    accuracy                           0.64       400
   macro avg       0.76      0.65      0.60       400
weighted avg       0.77      0.64      0.60       400
```



```
TRIGRAMS
Accuracy: 0.5

Classification Report:
               precision    recall  f1-score   support

    negative       0.49      0.99      0.65       191
    positive       0.85      0.05      0.10       209

    accuracy                           0.50       400
   macro avg       0.67      0.52      0.38       400
weighted avg       0.68      0.50      0.36       400

Accuracy: 0.5225

Classification Report:
               precision    recall  f1-score   support

    negative       0.50      0.98      0.66       191
    positive       0.88      0.10      0.18       209

    accuracy                           0.52       400
   macro avg       0.69      0.54      0.42       400
weighted avg       0.70      0.52      0.41       400

Accuracy: 0.5025

Classification Report:
               precision    recall  f1-score   support

    negative       0.49      0.99      0.66       191
    positive       0.86      0.06      0.11       209

    accuracy                           0.50       400
   macro avg       0.67      0.52      0.38       400
weighted avg       0.68      0.50      0.37       400
```

Figure 6: Evaluation Metrics of different n-grams values

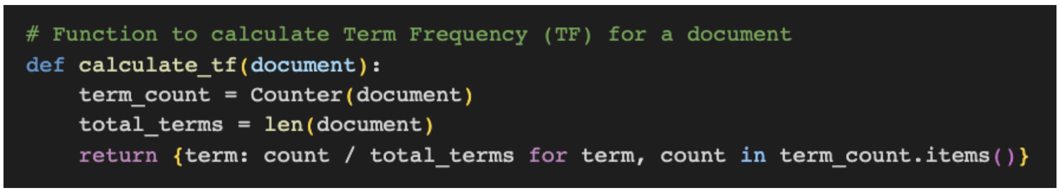
3.4 Normalisation

In the normalisation phase of the code, I created three different normalisation functions to be used on the feature sets, TF-IDF (Term Frequency - Inverse Document Frequency), Frequency Normalisation, and PPMI. It is important to normalise the datasets as the scales of term frequencies across different documents and datasets vary. The different normalisation techniques ensure the adaptability and effectiveness of the models I am evaluating. I use the `Counter` function imported from `collections`.

3.4.1 TF-IDF

TF-IDF is one of the normalisation techniques tested on the feature sets. The TF-IDF values are calculated to normalise the importance of terms within each document by considering both the frequency of a term in a document (TF) and its rarity across the entire dataset (IDF).

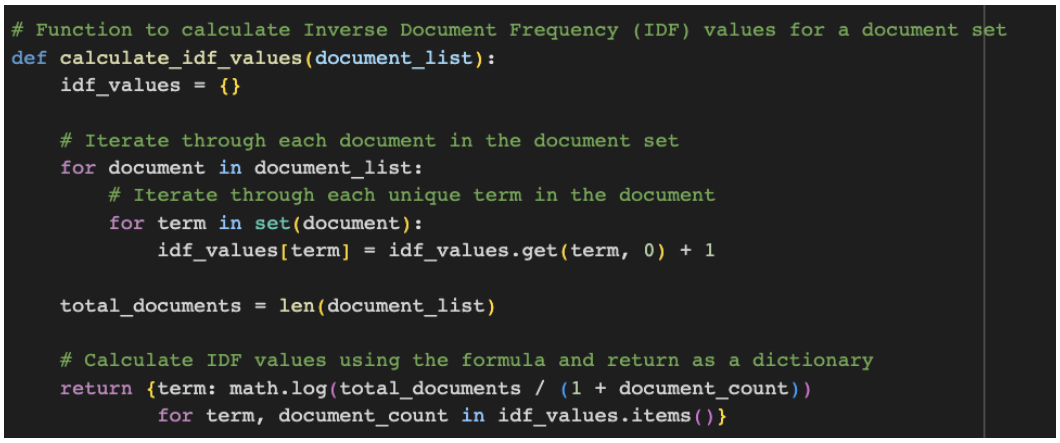
TF captures the significance of a term within a particular document. In my code, I calculated the TF value for each document, representing the proportion of each term's occurrence relative to the total terms in that document. This helps get rid of the influence of document length on term frequencies.



```
# Function to calculate Term Frequency (TF) for a document
def calculate_tf(document):
    term_count = Counter(document)
    total_terms = len(document)
    return {term: count / total_terms for term, count in term_count.items()}
```

Figure 7: Term Frequency Code

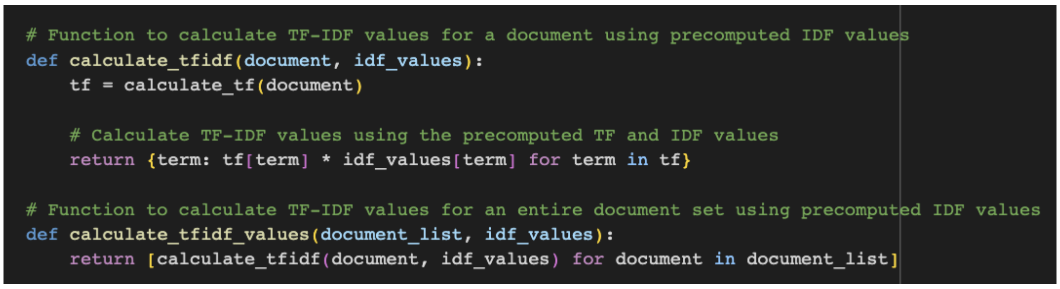
IDF addresses the issue of terms that are overly common across the entire dataset. Words that occur many times may not contribute much information. In my code, I calculated the IDF value by taking the inverse of the document frequency of each term. A logarithmic transformation is used to highlight the importance of rarer terms, thus giving those terms more weight.



```python
# Function to calculate Inverse Document Frequency (IDF) values for a document set
def calculate_idf_values(document_list):
    idf_values = {}

    # Iterate through each document in the document set
    for document in document_list:
        # Iterate through each unique term in the document
        for term in set(document):
            idf_values[term] = idf_values.get(term, 0) + 1

    total_documents = len(document_list)

    # Calculate IDF values using the formula and return as a dictionary
    return {term: math.log(total_documents / (1 + document_count))
            for term, document_count in idf_values.items()}
```

Figure 8: Inverse Document Frequency Code

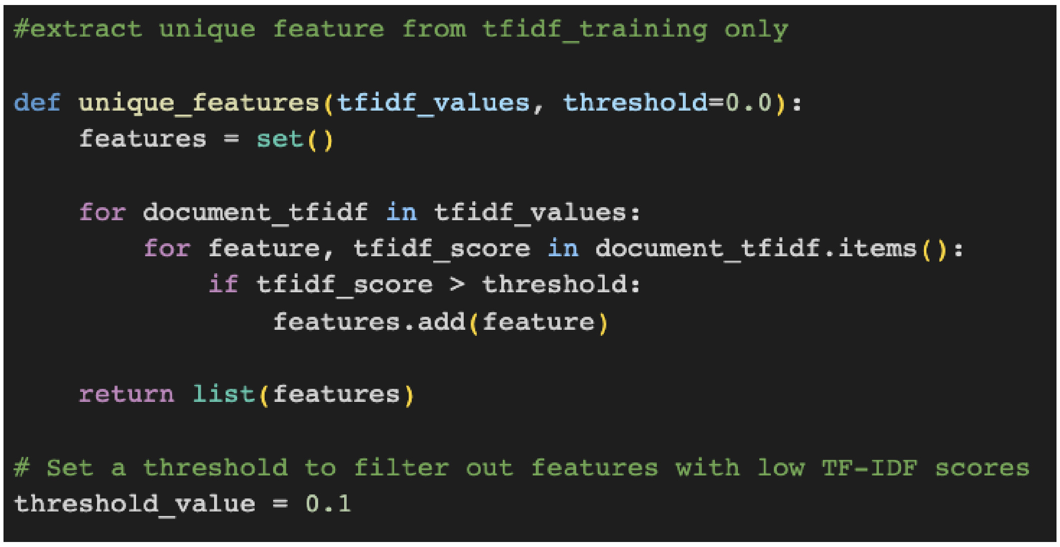
TF-IDF provides a normalized representation that balances the local importance of a term within a document with its global significance across the dataset. In my code, I calculated the TF-IDF value for each document by multiplying the TF and IDF values for each term in the document. This ensures that terms with high frequencies in a specific document but low occurrence across the dataset receive an appropriate importance level. Using that TF-IDF calculation function, I then calculated the TF-IDF value for all documents in each dataset – training, development and testing – separately using the function `calculate_tfidf_values`. This normalises each dataset separately which allows for individualised representations based on the unique features and vocabulary of the data.



```python
# Function to calculate TF-IDF values for a document using precomputed IDF values
def calculate_tfidf(document, idf_values):
    tf = calculate_tf(document)

    # Calculate TF-IDF values using the precomputed TF and IDF values
    return {term: tf[term] * idf_values[term] for term in tf}

# Function to calculate TF-IDF values for an entire document set using precomputed IDF values
def calculate_tfidf_values(document_list, idf_values):
    return [calculate_tfidf(document, idf_values) for document in document_list]
```

Figure 9: TF-IDF Code

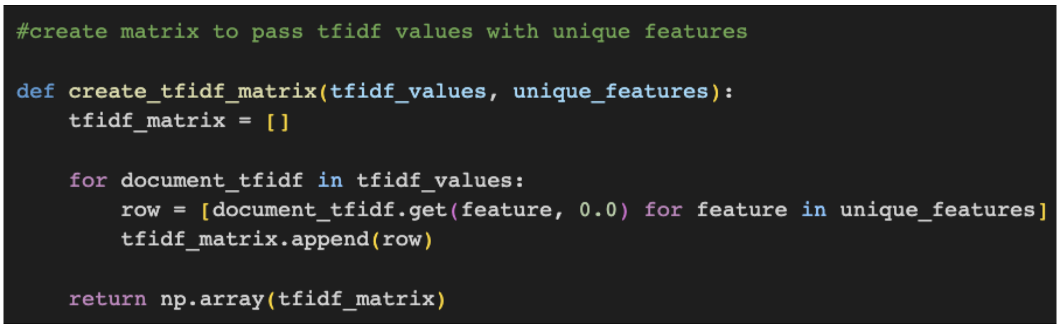
To create a TF-IDF matrix I have to first extract the unique features. The function `unique_features` is applied to the TF-IDF values of the training set above a specified threshold. By setting a threshold, it focuses on the terms that have a relatively higher impact on document representation. Extracting unique features from the training set ensures that the model is exposed only to the most relevant and informative terms for the task at hand. If features are extracted from the development or testing sets, it would potentially introduce terms that the model has not seen during training, leading to a mismatch between the training and evaluation phases. The purpose of extracting unique features is to filter out terms that do not contribute significantly to the content of the documents.



```
#extract unique feature from tfidf_training only

def unique_features(tfidf_values, threshold=0.0):
    features = set()

    for document_tfidf in tfidf_values:
        for feature, tfidf_score in document_tfidf.items():
            if tfidf_score > threshold:
                features.add(feature)

    return list(features)

# Set a threshold to filter out features with low TF-IDF scores
threshold_value = 0.1
```

Figure 10: Creating unique features code

The function `create_tfidf_matrix` converts the TF-IDF values of each document into a matrix format. Each row in the matrix corresponds to a document in the respective dataset – training, development, or testing – and each column corresponds to a unique feature extracted from the training set. The values in the matrix correspond to the TF-IDF scores associated with each unique feature in the respective document. If a particular feature is not present in a document or has a TF-IDF score below the specified threshold, the entry is set to 0.0.



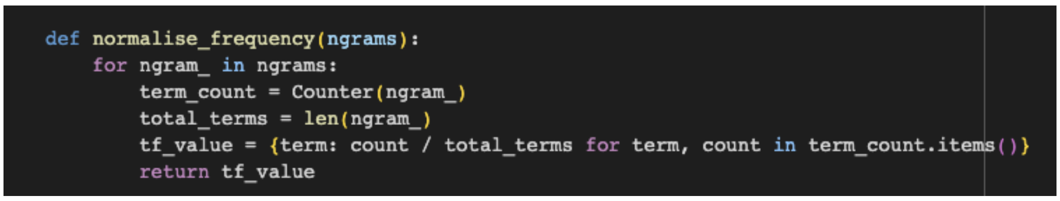
```
#create matrix to pass tfidf values with unique features

def create_tfidf_matrix(tfidf_values, unique_features):
    tfidf_matrix = []

    for document_tfidf in tfidf_values:
        row = [document_tfidf.get(feature, 0.0) for feature in unique_features]
        tfidf_matrix.append(row)

    return np.array(tfidf_matrix)
```

Figure 11: Creating TF-IDF matrix

3.4.2 Frequency Normalisation

Frequency normalisation is another technique employed to process and enhance the representation of text data in NLP. In this method, the number of times a term appears in a document is counted. Normalising frequencies is important because longer documents might have higher term frequencies, which may impose a bias on the importance of the term.

In the context of my code, I implemented frequency normalisation by dividing the raw term frequency (TF) of each term in a document by the total number of terms in that document. The aim of this process is to ensure that the importance of a term takes into account its proportional importance within the context of a particular document.



```python
def normalise_frequency(ngrams):
    for ngram_ in ngrams:
        term_count = Counter(ngram_)
        total_terms = len(ngram_)
        tf_value = {term: count / total_terms for term, count in term_count.items()}
        return tf_value
```

Figure 12: Frequency Normalisation Code

Through experiments, it was evident that frequency normalisation did not perform as well as TF-IDF. Therefore, for the rest of my project, I used TF-IDF normalisation techniques in order to achieve the best accuracy.

3.4.2 PPMI (Positive Pointwise Mutual Information)

PPMI is also another normalisation technique, it is used to evaluate the association between terms in a document. It specifically focuses on the positive correlation between the occurrences of two terms, eliminating any random chance associations. PPMI is beneficial in capturing meaningful relationships between terms. Another advantage it has is filtering out noise in the data.

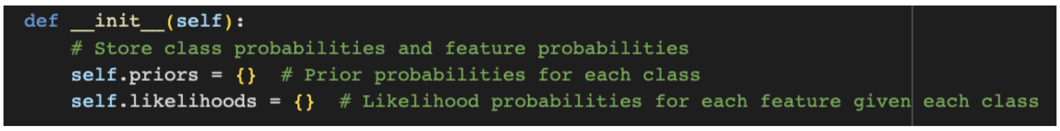
Unfortunately, I didn't have enough time to implement the PPMI normalisation technique. If I did have time to code it, I would start by applying the PPMI transformation to a term frequency (TF) matrix. I would then calculate the PPMI for a given pair of terms by comparing their co-occurrence probability with their individual probabilities. This formula aims to allocate higher scores to pairs of terms that have a higher frequency of co-occurrence than would be anticipated by random chance.

3.5 Naïve Bayes

Naïve Bayes is a probabilistic classification algorithm based on Bayes' theorem. This theorem uses prior knowledge of conditions that might relate to the event to calculate the probability of the event happening. In the context of text classification, Naïve Bayes is commonly used to classify documents into predefined categories, specifically for my project, into positive and negative. In this section I implemented and evaluated a Naïve Bayes classifier developed from scratch, not using any in-built functions.

3.5.1 Implementation

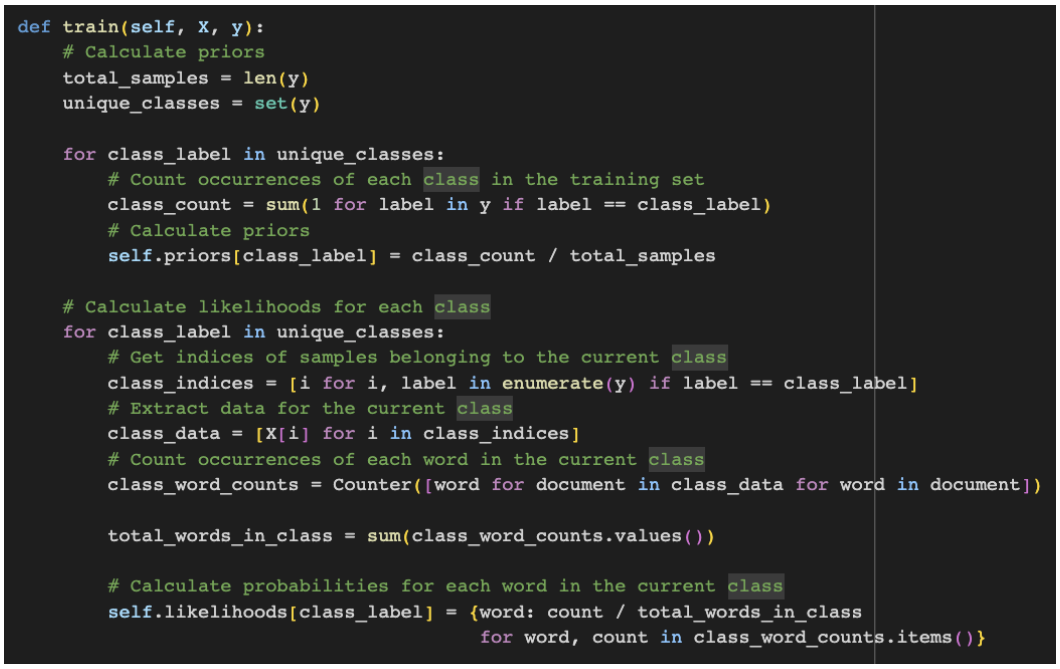
I created a class called, `NaiveBayesClassifier` which initialises two dictionaries, `self.priors` to store prior probabilities, and `self.likelihoods` to store the likelihood probabilities.



```
def __init__(self):
    # Store class probabilities and feature probabilities
    self.priors = {}  # Prior probabilities for each class
    self.likelihoods = {}  # Likelihood probabilities for each feature given each class
```

Figure 13: Initialisation

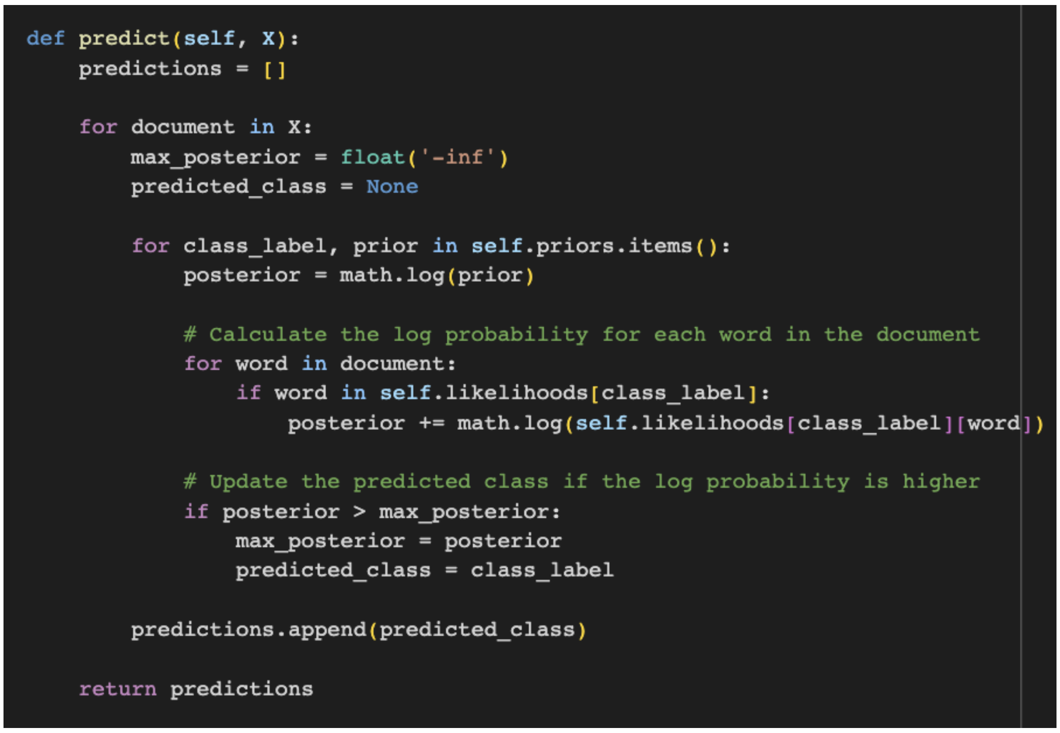
In the `train` method, calculates the prior probabilities and likelihood probabilities based on the training data, `X`, and the corresponding labels `y`. Prior probabilities are calculated by counting how many times each class occurs and dividing it by the total number of samples. Likelihood probabilities are calculated by counting how many times each feature in each class occurs and then dividing it by the total number of words in that class.



```python
def train(self, X, y):
    # Calculate priors
    total_samples = len(y)
    unique_classes = set(y)

    for class_label in unique_classes:
        # Count occurrences of each class in the training set
        class_count = sum(1 for label in y if label == class_label)
        # Calculate priors
        self.priors[class_label] = class_count / total_samples

    # Calculate likelihoods for each class
    for class_label in unique_classes:
        # Get indices of samples belonging to the current class
        class_indices = [i for i, label in enumerate(y) if label == class_label]
        # Extract data for the current class
        class_data = [X[i] for i in class_indices]
        # Count occurrences of each word in the current class
        class_word_counts = Counter([word for document in class_data for word in document])

        total_words_in_class = sum(class_word_counts.values())

        # Calculate probabilities for each word in the current class
        self.likelihoods[class_label] = {word: count / total_words_in_class
                                         for word, count in class_word_counts.items()}
```

Figure 14: Training

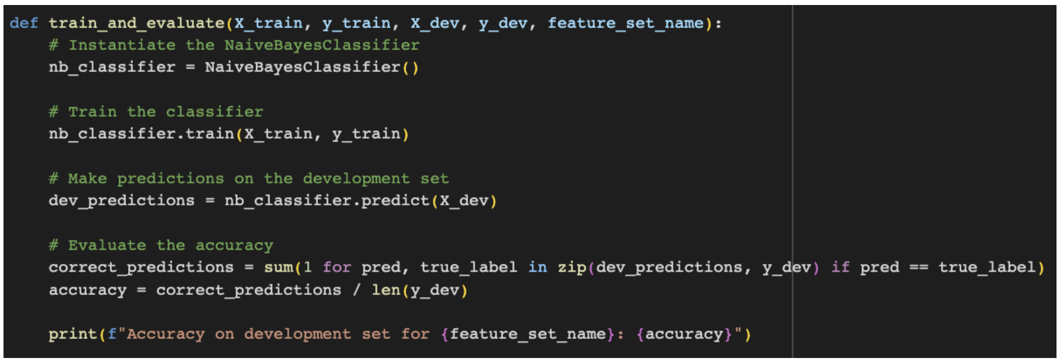
The `predict` method takes a list of documents, `X`, and predicts the class labels for each document based on the trained model. Prediction is done on the evaluation set.



```python
def predict(self, X):
    predictions = []

    for document in X:
        max_posterior = float('-inf')
        predicted_class = None

        for class_label, prior in self.priors.items():
            posterior = math.log(prior)

            # Calculate the log probability for each word in the document
            for word in document:
                if word in self.likelihoods[class_label]:
                    posterior += math.log(self.likelihoods[class_label][word])

            # Update the predicted class if the log probability is higher
            if posterior > max_posterior:
                max_posterior = posterior
                predicted_class = class_label

        predictions.append(predicted_class)

    return predictions
```

Figure 15: Predicting

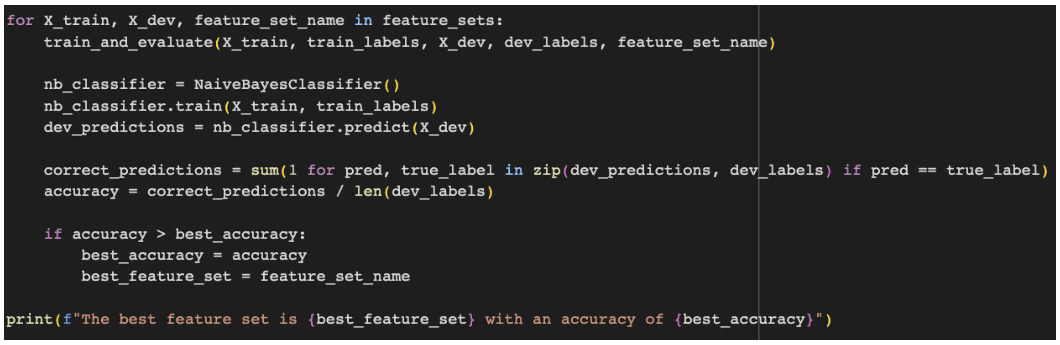
The `train_and_evaluate` function is used to set up the `NaiveBayesClassifier`. The code iterates over the three different feature sets, training the Naïve Bayes classifier on the training set, making predictions on the development set, and calculating the accuracy of those predictions.



```python
def train_and_evaluate(X_train, y_train, X_dev, y_dev, feature_set_name):
    # Instantiate the NaiveBayesClassifier
    nb_classifier = NaiveBayesClassifier()

    # Train the classifier
    nb_classifier.train(X_train, y_train)

    # Make predictions on the development set
    dev_predictions = nb_classifier.predict(X_dev)

    # Evaluate the accuracy
    correct_predictions = sum(1 for pred, true_label in zip(dev_predictions, y_dev) if pred == true_label)
    accuracy = correct_predictions / len(y_dev)

    print(f"Accuracy on development set for {feature_set_name}: {accuracy}")
```

Figure 16: Code for the `train_and_evaluate` function

I then compare the accuracy of all the feature sets, and the feature set with the highest accuracy is then identified as the best-performing one. The final output includes a summary that indicates the best feature set and its corresponding accuracy.

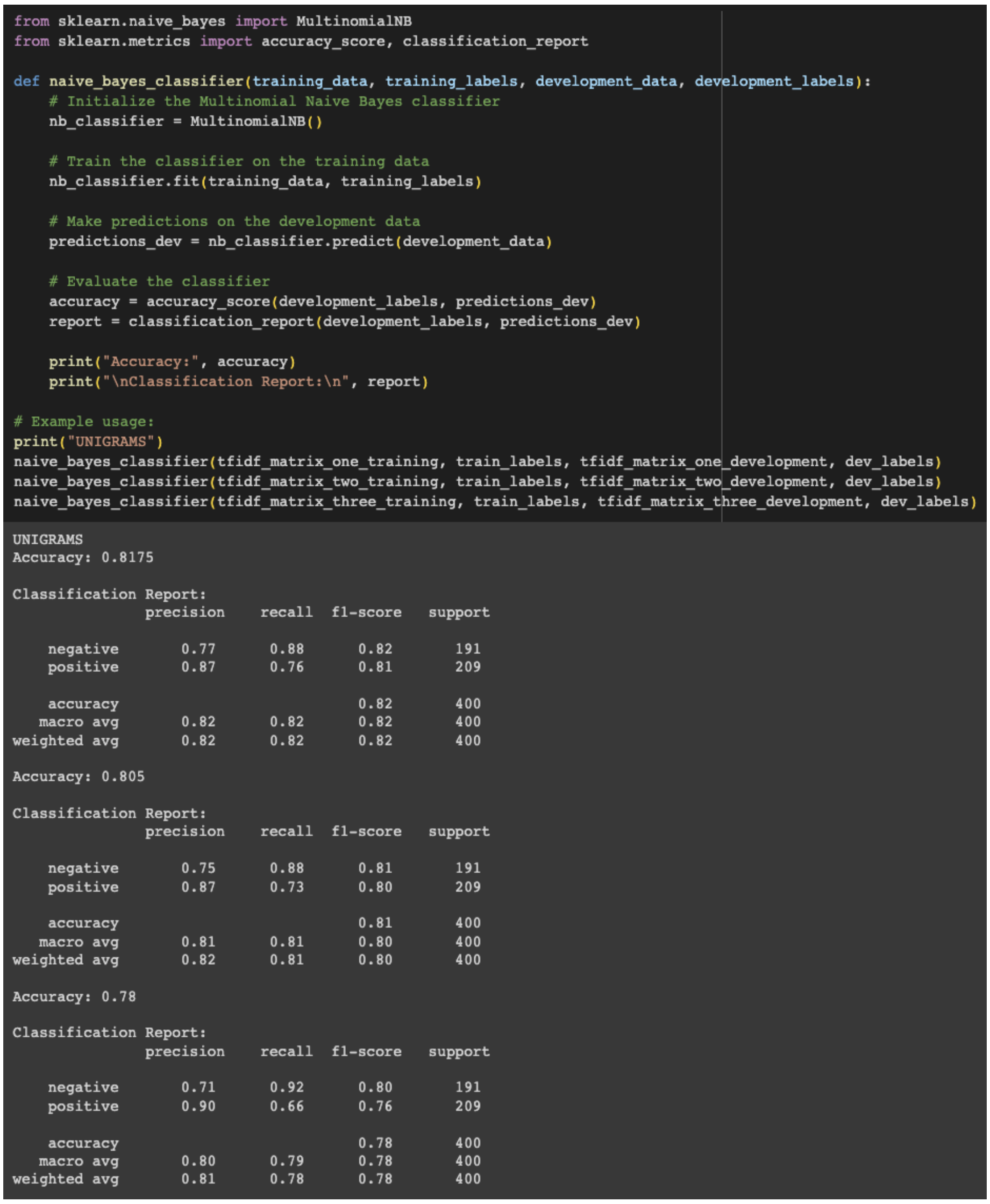


```python
for X_train, X_dev, feature_set_name in feature_sets:
    train_and_evaluate(X_train, train_labels, X_dev, dev_labels, feature_set_name)

    nb_classifier = NaiveBayesClassifier()
    nb_classifier.train(X_train, train_labels)
    dev_predictions = nb_classifier.predict(X_dev)

    correct_predictions = sum(1 for pred, true_label in zip(dev_predictions, dev_labels) if pred == true_label)
    accuracy = correct_predictions / len(dev_labels)

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_feature_set = feature_set_name

print(f"The best feature set is {best_feature_set} with an accuracy of {best_accuracy}")
```

Figure 17: Comparing accuracies between feature sets

3.5.2 Results

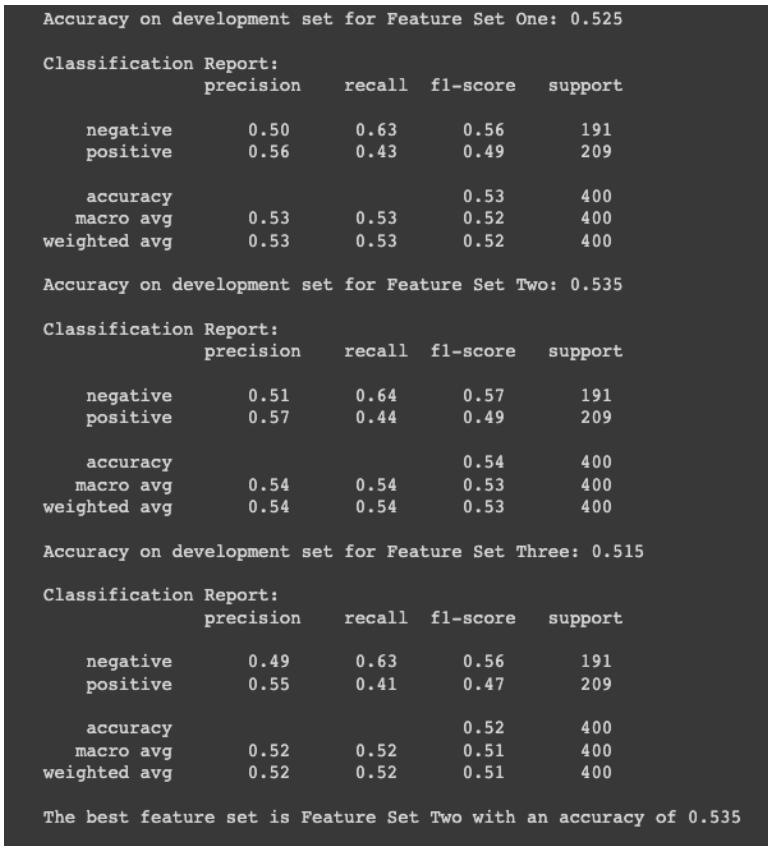
The three feature sets are evaluated by the development set on my own implementation as well as on the MultinomialNB implementation. As the figures below show, unfortunately, my own implementation of Naïve Bayes doesn't perform as well as the in-built implementation. Additionally, on my own implementation, feature set 2 performs the best with the highest accuracy. However, with the in-built implementation, feature set 1 does the best.



```python
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report

def naive_bayes_classifier(training_data, training_labels, development_data, development_labels):
    # Initialize the Multinomial Naive Bayes classifier
    nb_classifier = MultinomialNB()

    # Train the classifier on the training data
    nb_classifier.fit(training_data, training_labels)

    # Make predictions on the development data
    predictions_dev = nb_classifier.predict(development_data)

    # Evaluate the classifier
    accuracy = accuracy_score(development_labels, predictions_dev)
    report = classification_report(development_labels, predictions_dev)

    print("Accuracy:", accuracy)
    print("\nClassification Report:\n", report)

# Example usage:
print("UNIGRAMS")
naive_bayes_classifier(tfidf_matrix_one_training, train_labels, tfidf_matrix_one_development, dev_labels)
naive_bayes_classifier(tfidf_matrix_two_training, train_labels, tfidf_matrix_two_development, dev_labels)
naive_bayes_classifier(tfidf_matrix_three_training, train_labels, tfidf_matrix_three_development, dev_labels)
```

```
UNIGRAMS
Accuracy: 0.8175

Classification Report:
               precision    recall  f1-score   support

    negative       0.77      0.88      0.82       191
    positive       0.87      0.76      0.81       209

    accuracy                           0.82       400
   macro avg       0.82      0.82      0.82       400
weighted avg       0.82      0.82      0.82       400

Accuracy: 0.805

Classification Report:
               precision    recall  f1-score   support

    negative       0.75      0.88      0.81       191
    positive       0.87      0.73      0.80       209

    accuracy                           0.81       400
   macro avg       0.81      0.81      0.80       400
weighted avg       0.82      0.81      0.80       400

Accuracy: 0.78

Classification Report:
               precision    recall  f1-score   support

    negative       0.71      0.92      0.80       191
    positive       0.90      0.66      0.76       209

    accuracy                           0.78       400
   macro avg       0.80      0.79      0.78       400
weighted avg       0.81      0.78      0.78       400
```

Figure 19: Code and Evaluation Metrics of in-built implementation of Naïve Bayes



```
Accuracy on development set for Feature Set One: 0.525

Classification Report:
              precision    recall  f1-score   support

    negative       0.50      0.63      0.56       191
    positive       0.56      0.43      0.49       209

    accuracy                           0.53       400
   macro avg       0.53      0.53      0.52       400
weighted avg       0.53      0.53      0.52       400

Accuracy on development set for Feature Set Two: 0.535

Classification Report:
              precision    recall  f1-score   support

    negative       0.51      0.64      0.57       191
    positive       0.57      0.44      0.49       209

    accuracy                           0.54       400
   macro avg       0.54      0.54      0.53       400
weighted avg       0.54      0.54      0.53       400

Accuracy on development set for Feature Set Three: 0.515

Classification Report:
              precision    recall  f1-score   support

    negative       0.49      0.63      0.56       191
    positive       0.55      0.41      0.47       209

    accuracy                           0.52       400
   macro avg       0.52      0.52      0.51       400
weighted avg       0.52      0.52      0.51       400

The best feature set is Feature Set Two with an accuracy of 0.535
```

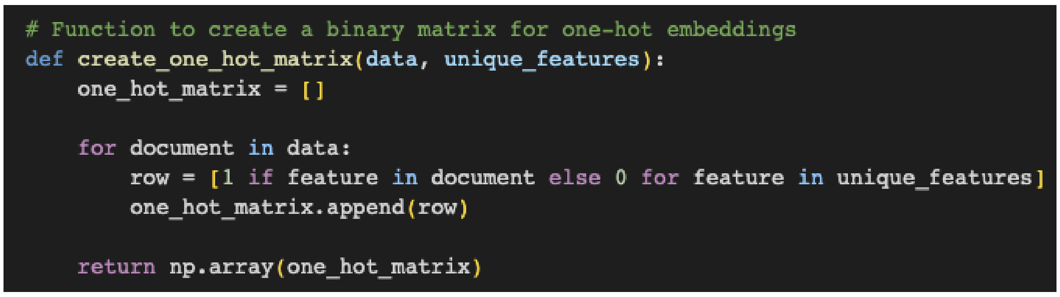
Figure 18: Evaluation Metrics of my own implementation of Naïve Bayes

3.6 SGD-based Classification and SVMs

In this phase of my project, I evaluated two distinct classification models, Stochastic Gradient Descent (SGD)-based classification and Support Vector Machines (SVMs). I used the `scikit-learn` library and imported Logistic Regression and SVM packages to evaluate the development set. The evaluation focuses on the three feature sets generated before, each extracted using unique techniques: `feature_selection_one`, `feature_selection_two`, and `feature_selection_three`.

3.6.1 Implementation

The function `create_one_hot_matrix` produces 1-hot embeddings based on the features which will be used as the input features for logistic regression and SVM models. `create_one_hot_matrix` takes two parameters, `data` (the feature set (a list of documents, where each document is represented as a list of features)) and `unique_features` (a list of unique features extracted from the training data). The function creates a binary matrix where each row corresponds to a document, and each column corresponds to a unique feature. If a feature is present in a document, the corresponding entry in the matrix is set to 1; otherwise, it's set to 0. 1-hot embeddings is a common method used to convert text data into a format that machine learning models can process.



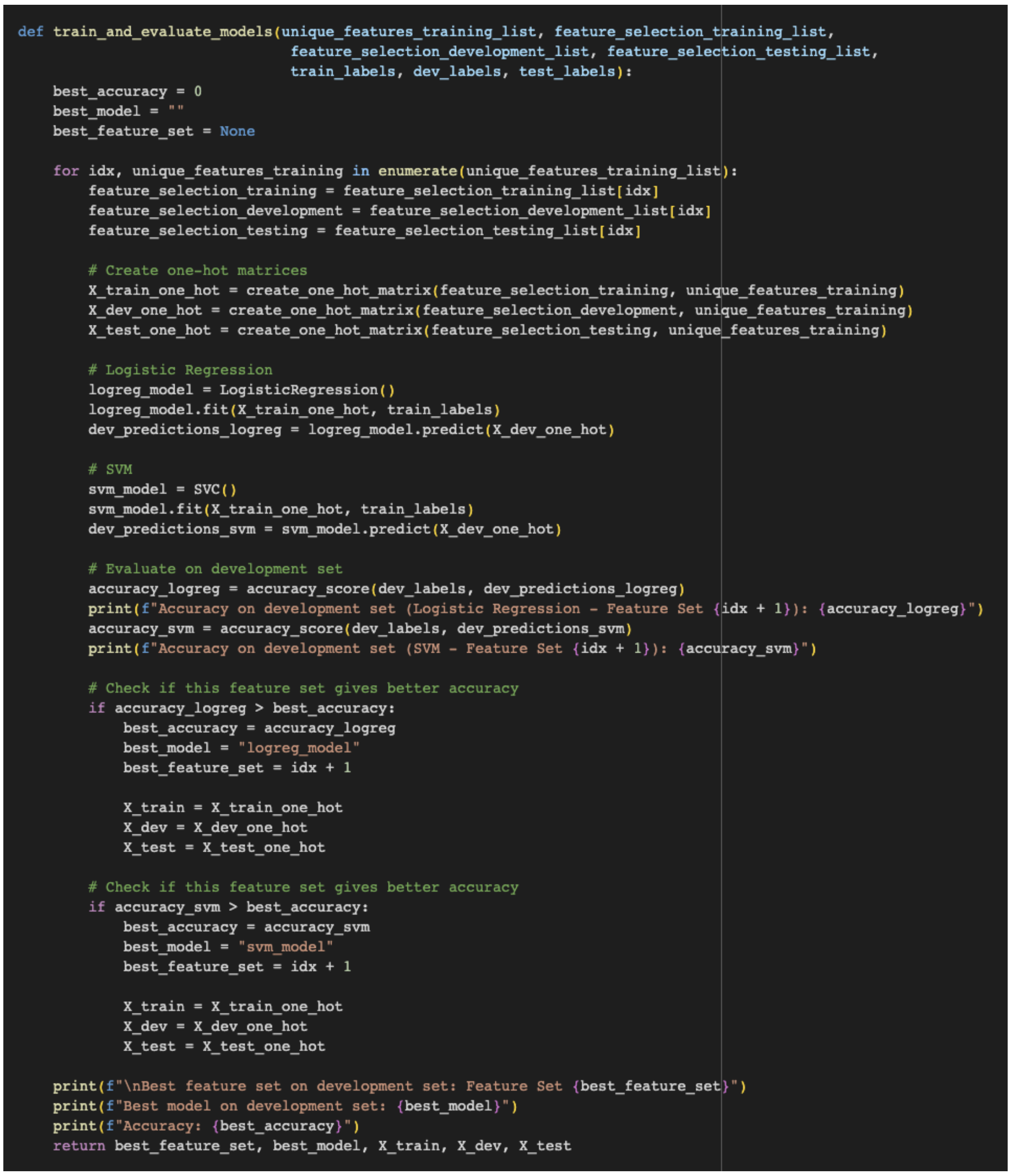
```python
# Function to create a binary matrix for one-hot embeddings
def create_one_hot_matrix(data, unique_features):
    one_hot_matrix = []

    for document in data:
        row = [1 if feature in document else 0 for feature in unique_features]
        one_hot_matrix.append(row)

    return np.array(one_hot_matrix)
```

Figure 19: Code for one hot embeddings

`logreg_model` represents a logistic regression model, `LogisticRegression()`, while `svm_model` represents a Support Vector Classification (SVC) model, `SVC()`. Both models are trained for binary classification tasks, where the goal is to predict one of two possible outcomes.

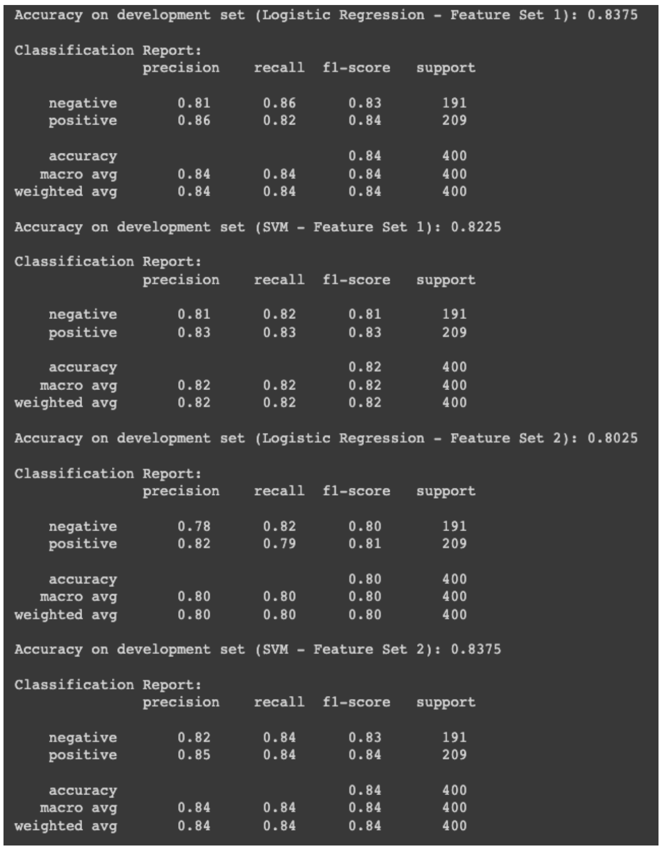
The `train_and_evaluate_model` function is used on all feature sets to find the feature set that performs with the best accuracy rate. Both classifiers are trained using the training set, `X_train_one_hot`, and labels, `train_labels`. Predictions for both models are then made on the development set, `X_dev_one_hot`. The predictions along with the corresponding labels, `dev_labels`, are then used to evaluate the accuracy of the model on the development set. This is shown in Figure 21 in the results section.

3.6.2 Results

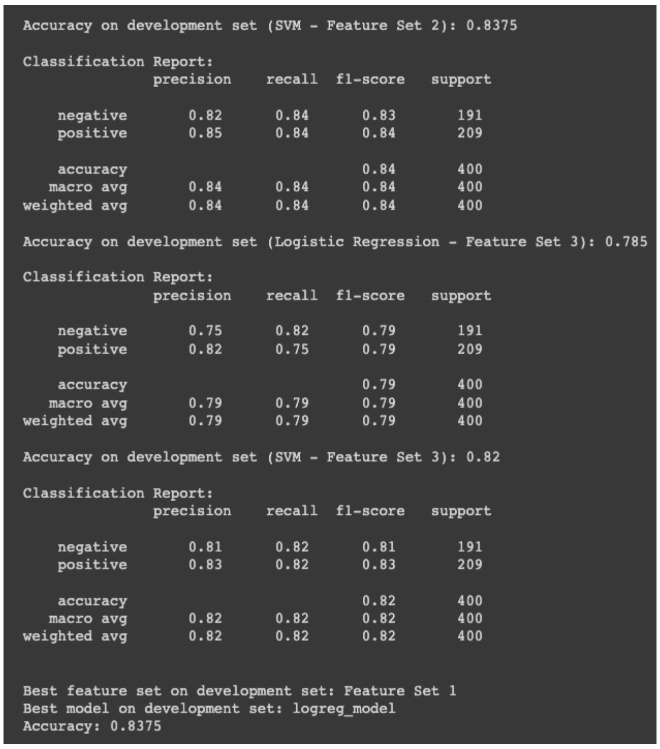


```python
def train_and_evaluate_models(unique_features_training_list, feature_selection_training_list,
                              feature_selection_development_list, feature_selection_testing_list,
                              train_labels, dev_labels, test_labels):
    best_accuracy = 0
    best_model = ""
    best_feature_set = None

    for idx, unique_features_training in enumerate(unique_features_training_list):
        feature_selection_training = feature_selection_training_list[idx]
        feature_selection_development = feature_selection_development_list[idx]
        feature_selection_testing = feature_selection_testing_list[idx]

        # Create one-hot matrices
        X_train_one_hot = create_one_hot_matrix(feature_selection_training, unique_features_training)
        X_dev_one_hot = create_one_hot_matrix(feature_selection_development, unique_features_training)
        X_test_one_hot = create_one_hot_matrix(feature_selection_testing, unique_features_training)

        # Logistic Regression
        logreg_model = LogisticRegression()
        logreg_model.fit(X_train_one_hot, train_labels)
        dev_predictions_logreg = logreg_model.predict(X_dev_one_hot)

        # SVM
        svm_model = SVC()
        svm_model.fit(X_train_one_hot, train_labels)
        dev_predictions_svm = svm_model.predict(X_dev_one_hot)

        # Evaluate on development set
        accuracy_logreg = accuracy_score(dev_labels, dev_predictions_logreg)
        print(f"Accuracy on development set (Logistic Regression - Feature Set {idx + 1}): {accuracy_logreg}")
        accuracy_svm = accuracy_score(dev_labels, dev_predictions_svm)
        print(f"Accuracy on development set (SVM - Feature Set {idx + 1}): {accuracy_svm}")

        # Check if this feature set gives better accuracy
        if accuracy_logreg > best_accuracy:
            best_accuracy = accuracy_logreg
            best_model = "logreg_model"
            best_feature_set = idx + 1

            X_train = X_train_one_hot
            X_dev = X_dev_one_hot
            X_test = X_test_one_hot

        # Check if this feature set gives better accuracy
        if accuracy_svm > best_accuracy:
            best_accuracy = accuracy_svm
            best_model = "svm_model"
            best_feature_set = idx + 1

            X_train = X_train_one_hot
            X_dev = X_dev_one_hot
            X_test = X_test_one_hot

    print(f"\nBest feature set on development set: Feature Set {best_feature_set}")
    print(f"Best model on development set: {best_model}")
    print(f"Accuracy: {best_accuracy}")
    return best_feature_set, best_model, X_train, X_dev, X_test
```

Figure 20: Code for training and evaluating both models



```
Accuracy on development set (Logistic Regression - Feature Set 1): 0.8375

Classification Report:
              precision    recall  f1-score   support

    negative       0.81      0.86      0.83       191
    positive       0.86      0.82      0.84       209

    accuracy                           0.84       400
   macro avg       0.84      0.84      0.84       400
weighted avg       0.84      0.84      0.84       400

Accuracy on development set (SVM - Feature Set 1): 0.8225

Classification Report:
              precision    recall  f1-score   support

    negative       0.81      0.82      0.81       191
    positive       0.83      0.83      0.83       209

    accuracy                           0.82       400
   macro avg       0.82      0.82      0.82       400
weighted avg       0.82      0.82      0.82       400

Accuracy on development set (Logistic Regression - Feature Set 2): 0.8025

Classification Report:
              precision    recall  f1-score   support

    negative       0.78      0.82      0.80       191
    positive       0.82      0.79      0.81       209

    accuracy                           0.80       400
   macro avg       0.80      0.80      0.80       400
weighted avg       0.80      0.80      0.80       400

Accuracy on development set (SVM - Feature Set 2): 0.8375

Classification Report:
              precision    recall  f1-score   support

    negative       0.82      0.84      0.83       191
    positive       0.85      0.84      0.84       209

    accuracy                           0.84       400
   macro avg       0.84      0.84      0.84       400
weighted avg       0.84      0.84      0.84       400
```



```
Accuracy on development set (SVM - Feature Set 2): 0.8375

Classification Report:
              precision    recall  f1-score   support

    negative       0.82      0.84      0.83       191
    positive       0.85      0.84      0.84       209

    accuracy                           0.84       400
   macro avg       0.84      0.84      0.84       400
weighted avg       0.84      0.84      0.84       400

Accuracy on development set (Logistic Regression - Feature Set 3): 0.785

Classification Report:
              precision    recall  f1-score   support

    negative       0.75      0.82      0.79       191
    positive       0.82      0.75      0.79       209

    accuracy                           0.79       400
   macro avg       0.79      0.79      0.79       400
weighted avg       0.79      0.79      0.79       400

Accuracy on development set (SVM - Feature Set 3): 0.82

Classification Report:
              precision    recall  f1-score   support

    negative       0.81      0.82      0.81       191
    positive       0.83      0.82      0.83       209

    accuracy                           0.82       400
   macro avg       0.82      0.82      0.82       400
weighted avg       0.82      0.82      0.82       400


Best feature set on development set: Feature Set 1
Best model on development set: logreg_model
Accuracy: 0.8375
```

Figure 21: Evaluation Metrics of all three feature sets and both models on the development set

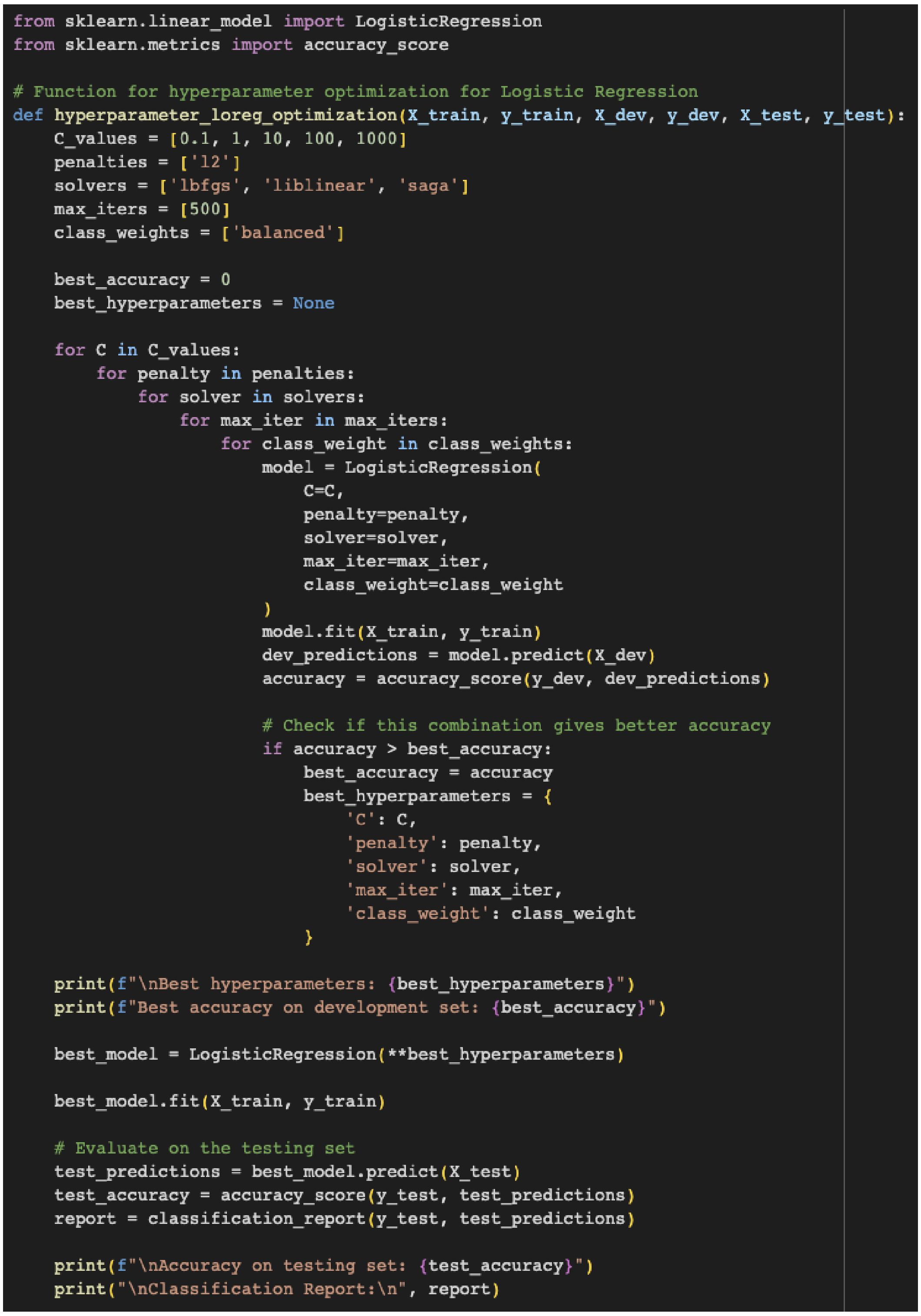
After comparing the accuracies across the different models and feature sets, I identified the best-performing feature set and model:

- Best feature set on development set: Feature Set 1
- Best model on development set: logreg_model
- Accuracy on development set: 0.8375

3.6.3 Hyperparameter Optimisations

The best-performing feature set was then selected to carry out further experiments. To optimise the performance of the selected feature set on both classifier models, I conducted hyperparameter optimisation. At least 5 different combinations of hyperparameters were evaluated on the feature set and models using the development set. For Logistic Regression, I explored these parameters:

- C (Regularisation Strength): controls the inverse of the regularization strength
    - Smaller values specify stronger regularization, preventing overfitting
- Penalty: specifies the norm of the penalty used in the regularisation term
    - 'l2' adds a L2 regularisation term, 'l1' adds a L1 regularisation term
- Solver: the algorithm used in the optimization problem
    - For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones
- Max Iterations: the maximum number of iterations taken for the solvers to converge.
- Class Weight: used to handle class imbalance by assigning different weights to classes.

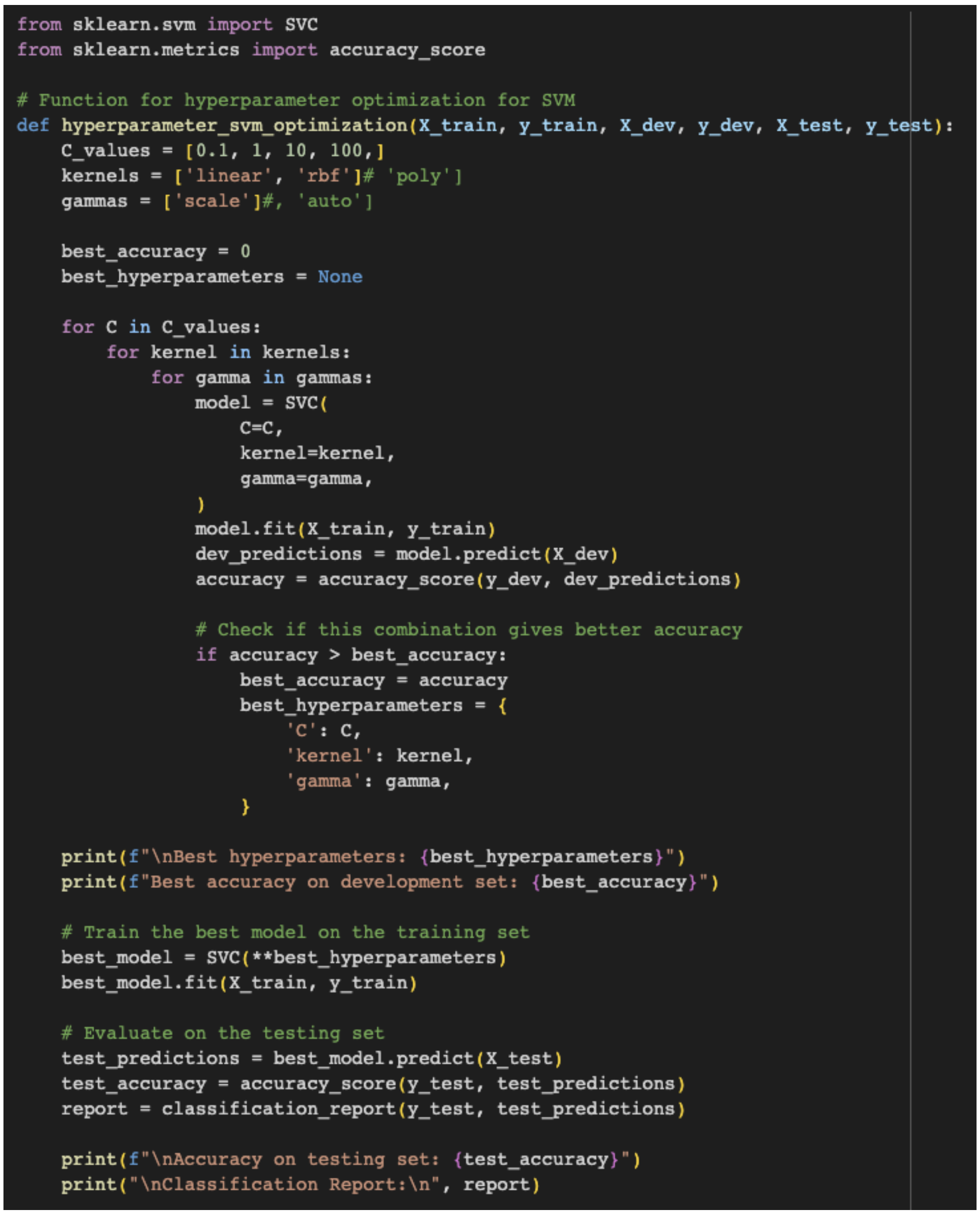


```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Function for hyperparameter optimization for Logistic Regression
def hyperparameter_loreg_optimization(X_train, y_train, X_dev, y_dev, X_test, y_test):
    C_values = [0.1, 1, 10, 100, 1000]
    penalties = ['l2']
    solvers = ['lbfgs', 'liblinear', 'saga']
    max_iters = [500]
    class_weights = ['balanced']

    best_accuracy = 0
    best_hyperparameters = None

    for C in C_values:
        for penalty in penalties:
            for solver in solvers:
                for max_iter in max_iters:
                    for class_weight in class_weights:
                        model = LogisticRegression(
                            C=C,
                            penalty=penalty,
                            solver=solver,
                            max_iter=max_iter,
                            class_weight=class_weight
                        )
                        model.fit(X_train, y_train)
                        dev_predictions = model.predict(X_dev)
                        accuracy = accuracy_score(y_dev, dev_predictions)

                        # Check if this combination gives better accuracy
                        if accuracy > best_accuracy:
                            best_accuracy = accuracy
                            best_hyperparameters = {
                                'C': C,
                                'penalty': penalty,
                                'solver': solver,
                                'max_iter': max_iter,
                                'class_weight': class_weight
                            }

    print(f"\nBest hyperparameters: {best_hyperparameters}")
    print(f"Best accuracy on development set: {best_accuracy}")

    best_model = LogisticRegression(**best_hyperparameters)

    best_model.fit(X_train, y_train)

    # Evaluate on the testing set
    test_predictions = best_model.predict(X_test)
    test_accuracy = accuracy_score(y_test, test_predictions)
    report = classification_report(y_test, test_predictions)

    print(f"\nAccuracy on testing set: {test_accuracy}")
    print("\nClassification Report:\n", report)
```

Figure 22: Hyperparameter Optimisation for Logistic Regression

For SVM, I explored these parameters:

- C (Regularization Strength): controls the trade-off between smooth decision boundaries and classifying training points correctly
    - The strength of the regularization is inversely proportional to C
- Kernel: specifies the kernel type to be used in the algorithm
- Gamma: defines how far the influence of a single training example reaches
    - Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.



```python
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Function for hyperparameter optimization for SVM
def hyperparameter_svm_optimization(X_train, y_train, X_dev, y_dev, X_test, y_test):
    C_values = [0.1, 1, 10, 100,]
    kernels = ['linear', 'rbf']# 'poly']
    gammas = ['scale']#, 'auto']

    best_accuracy = 0
    best_hyperparameters = None

    for C in C_values:
        for kernel in kernels:
            for gamma in gammas:
                model = SVC(
                    C=C,
                    kernel=kernel,
                    gamma=gamma,
                )
                model.fit(X_train, y_train)
                dev_predictions = model.predict(X_dev)
                accuracy = accuracy_score(y_dev, dev_predictions)

                # Check if this combination gives better accuracy
                if accuracy > best_accuracy:
                    best_accuracy = accuracy
                    best_hyperparameters = {
                        'C': C,
                        'kernel': kernel,
                        'gamma': gamma,
                    }

    print(f"\nBest hyperparameters: {best_hyperparameters}")
    print(f"Best accuracy on development set: {best_accuracy}")

    # Train the best model on the training set
    best_model = SVC(**best_hyperparameters)
    best_model.fit(X_train, y_train)

    # Evaluate on the testing set
    test_predictions = best_model.predict(X_test)
    test_accuracy = accuracy_score(y_test, test_predictions)
    report = classification_report(y_test, test_predictions)

    print(f"\nAccuracy on testing set: {test_accuracy}")
    print("\nClassification Report:\n", report)
```

Figure 23: Hyperparameter Optimisation for SVM

After evaluating the development set, the best hyperparameters are then chosen. Both models are then trained again on the training set with the best-performing hyperparameters, `**best_hyperparameters`. Lastly, predictions on the test set, `X_test`, are made, which is called `test_predictions`, and the accuracy is calculated by evaluating the test set, `y_test`, using the predictions made. Further evaluation metrics are also calculated using `classification_report()`. The evaluation results on the test set is shown in the Results for Hyperparameters optimisation section.

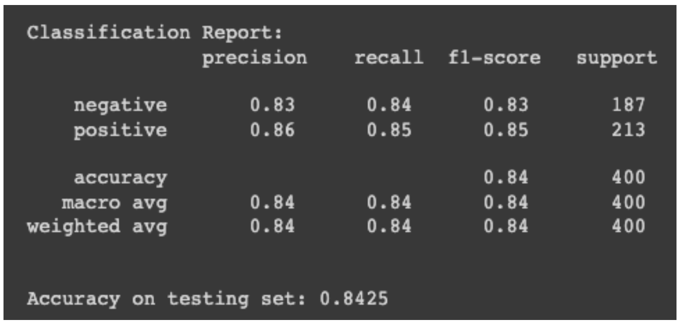
3.6.2 Results for Hyperparameters optimisation

The best hyperparameters for the Logistic Regression model are as follows:

- Best hyperparameters: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs', 'max_iter': 500, 'class_weight': 'balanced'}
- Best accuracy on development set: 0.8375

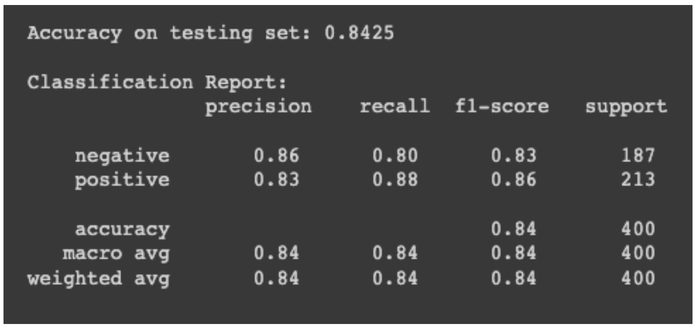
The best hyperparameters for the SVM model are as follows:

- Best hyperparameters: {'C': 1, 'kernel': 'rbf', 'gamma': 'scale'}
- Best accuracy on development set: 0.8225



```
Classification Report:
              precision    recall  f1-score   support

    negative       0.83      0.84      0.83       187
    positive       0.86      0.85      0.85       213

    accuracy                           0.84       400
   macro avg       0.84      0.84      0.84       400
weighted avg       0.84      0.84      0.84       400


Accuracy on testing set: 0.8425
```

Figure 24: Evaluation Metrics of Hyperparametr Optimisation for Logistic Regression



```
Accuracy on testing set: 0.8425

Classification Report:
              precision    recall  f1-score   support

    negative       0.86      0.80      0.83       187
    positive       0.83      0.88      0.86       213

    accuracy                           0.84       400
   macro avg       0.84      0.84      0.84       400
weighted avg       0.84      0.84      0.84       400
```

Figure 25: Evaluation Metrics of Hyperparametr Optimisation for SVM

The code from this section demonstrates how to use one-hot embeddings based on the extracted features, train logistic regression and SVM models. Evaluate them on the development set, and select the best-performing model for evaluation on the test set.

3.7 BERT (Bidirectional Encoder Representations from Transformers)

BERT is a state-of-the-art pre-trained NLP model. It belongs to the transformer-based models, a type of neural network architecture designed to handle sequential data, such as text. BERT is well known for its bidirectional context understanding, enabling it to consider both the left and right contexts of words, providing a better understanding of language. BERT is pre-trained on large amounts of text data, allowing the model to capture complex linguistic patterns, semantic meanings, and relationships between words.

Unfortunately, I didn't have enough time to implement BERT. However, from the research I have done on the model, I have a vague idea of how I would implement it if I had extra time. Using the same training, development and testing splits as before, I would implement BERT, specifically the base model, to perform the same classification task. Firstly, I would explore the custom datasets tutorial from https://huggingface.co/transformers/v3.2.0/custom_datasets.html, to understand how to format and load my custom dataset, to ensure that it is compatible with the BERT model. Next, I would refer to the IMDB Classification tutorial (https://huggingface.co/docs/transformers/tasks/sequence_classification) to learn how to fine-tune BERT for the specific classification task using the custom data splits.

I would experiment with both of BERT's cased and uncased versions to observe their performance on the task. The cased version preserves case information, while the uncased version treats all text as lowercase. Therefore, the uncased version should capture more general patterns. By comparing the results of these experiments, I could determine which version of BERT is more suitable for the given classification task.

In summary, my approach would involve loading the custom dataset, fine-tuning the BERT base model on the provided train/dev/test splits, and experimenting with both cased and uncased versions to optimise performance for the specific text classification problem at hand.

4 Discussion

This section shows the performance of each model on the test set. All feature sets are evaluated and the metrics used include, precision, recall, f-1 score and accuracy.

Method	Feature Sets	Evaluation Metrics			
		Precision	Recall	F-1 Score	Accuracy
Naïve Bayes (own)	**1**	**0.50**	**0.50**	**0.49**	**0.49**
	2	0.49	0.49	0.48	0.48
	3	0.44	0.44	0.43	0.43
Naïve Bayes (scikit-learn)	**1**	**0.82**	**0.82**	**0.81**	**0.81**
	2	**0.82**	**0.82**	**0.81**	**0.81**
	3	0.82	0.80	0.79	0.79
SGD	**1**	**0.85**	**0.85**	**0.85**	**0.85**
	2	0.82	0.82	0.82	0.82
	3	0.78	0.78	0.78	0.78
SVM	**1**	**0.84**	**0.84**	**0.84**	**0.84**
	2	0.83	0.83	0.83	0.83
	3	0.79	0.79	0.79	0.79
SGD - Hyperparameter Optimisation	**1**	**0.84**	**0.84**	**0.84**	**0.84**
SVM - Hyperparameter Optimisation	**1**	**0.84**	**0.84**	**0.84**	**0.84**

4.1 Naïve Bayes (my implementation)

My implementation of the Naïve Bayes classifier performed the best on the first feature set, with an accuracy score of 49%. The second feature set performed almost as well as the first one, this makes sense as the first feature set has one additional feature. The model performed the worst on the third feature set. Therefore, it can be interpreted that Stemming performs better than Lemmatising for text classification.

4.2 Naïve Bayes (scikit-learn)

Whilst the `scikit-learn` Naïve Bayes achieved an accuracy of 81% for the first and second feature sets. Similar to my implementation, the third feature set did the worst. This implementation does significantly better compared to my implementation of Naïve Bayes as this has been optimised and extensively tested, ensuring robustness and accuracy. The library uses advanced methods to handle edge cases, smoothing, and feature scaling, all of which can extensively enhance the model's performance. Furthermore, Multinomial Naive Bayes is well-suited for text classification tasks involving discrete features like word frequencies.

4.3 SGD

To implement SGD, I used a Logistic Regression Model which has proved to be very effective. It achieved an accuracy of 85% on the first feature set. The worst feature set is the third one where it achieved 78%. SGD classifiers are known for their efficiency in optimising large-scale machine learning models. The variation in performance across feature sets may be attributed to the nature of the data and the model's sensitivity to certain features. This model outperforms Naïve Bayes in text classification due to its adaptability to feature representations and efficient optimisation with mini-batch updates. While Naïve Bayes relies on the assumption of feature independence, Logistic Regression demonstrates versatility and can excel in scenarios where datasets are large.

4.4 SVM

The SVM model did well across the feature sets, with an accuracy of 84% on the first feature set being the best performance. SVM is known for its effectiveness in handling high-dimensional datasets and finding optimal decision boundaries, making it suitable for text classification tasks. However, it is computationally expensive, taking a significantly longer time compared to Logistic Regression. Additionally, Logistic Regression is more flexible in capturing subtle relationships within the data.

4.5 Hyperparameter Optimisation (SGD and SVM)

Hyperparameter Optimisation although performs well, does not outperform the models before the hyperparameters were optimised which I was surprised to see. It has an accuracy of 84% for both Logistic Regression and SVM which is 1% less than the initial Logistic Regression model. I assumed that hyperparameter optimisation would have performed better than the initial models as for logistic regression, careful tuning of hyperparameters can impact the model's convergence and generalisation, enhancing the efficiency and effectiveness of the model. As for SVM, the choice of kernel and other parameters can also influence the model's performance. When evaluating the feature sets on the development models, both models achieved higher accuracies with hyperparameter optimisation. However, this is not the case with the test set.

4.6 BERT

Although I don't have the data for BERT, I believe the model would have outperformed these traditional models. BERT's ability to capture contextual information and understand complex language expressions would help improve the performance level. Additionally, considering the bidirectional context of words in a sentence gives the model an advantage compared to the others. This makes it particularly effective for tasks where context and semantic understanding are crucial, which is what I am evaluating.

5 Conclusion and Future Work

I have found the SGD-Classifier using Logistic Regression to be the best-performing model, achieving an accuracy of 85% on the first feature set. It outperformed Naïve Bayes, SVM, and hyperparameter-optimized versions of SGD and SVM. The best-performing feature set is the first feature set where stemming, lowercase conversion, removal of stop words, and removal of punctuation marks were used.

For future works, I intend to continue fine-tuning the hyperparameters for both Logistic Regression and SVM models. Furthermore, exploring and evaluating BERT can be beneficial, providing additional results to sentiment analysis.

Bibliography

Barney, N. (2023) What is sentiment analysis (opinion mining)?: Definition from TechTarget, Business Analytics. Available at: https://www.techtarget.com/searchbusinessanalytics/definition/opinion-mining-sentiment-mining.

Collomb, A. et al. (2014) Study and comparison of sentiment analysis methods for reputation ..., A Study and Comparison of Sentiment Analysis Methods for Reputation Evaluation. Available at: https://liris.cnrs.fr/Documents/Liris-6508.pdf.

Gupta, S. (2018) Sentiment analysis: Concept, analysis and applications, Medium. Available at: https://towardsdatascience.com/sentiment-analysis-concept-analysis-and-applications-6c94d6f58c17.

Kannan, S. et al. (2016) Lexicon-based approach, Lexicon-Based Approach - an overview | ScienceDirect Topics. Available at: https://www.sciencedirect.com/topics/computer-science/lexicon-based-approach.

Maas, A. (2011) Large Movie Review Dataset, Sentiment Analysis. Available at: https://ai.stanford.edu/~amaas/data/sentiment/.

Yılmaz, B. (2023) Sentiment Analysis Methods in 2023: Overview, pros &amp; cons, AIMultiple. Available at: https://research.aimultiple.com/sentiment-analysis-methods/ .

Yılmaz, B. (2023) Top 5 sentiment analysis challenges and solutions in 2023, AIMultiple. Available at: https://research.aimultiple.com/sentiment-analysis-challenges/.