PART A - Sentiment Analysis

Abstract

This paper dives into the world of sentiment analysis, digging into the task of understanding and interpreting the emotions expressed in any written work. The aim of this project is to develop an application that can detect the sentimental polarity given any input of text. Incorporating a set of reviews extracted from the Internet Movie Database(IMDb), we explored the different methods of feature generation and selection as well as implemented a machine-based learning technique (Naive Bayes) for sentiment analysis. To test the effectiveness of each method, various evaluation metrics were used. The results obtained from these methods demonstrated the diverse ways in which they contributed to sentiment analysis, shedding light on the most effective approach for accurate and insightful interpretation.

Keywords: sentimental analysis; feature selection; Naive Bayes

Introduction

Sentiment analysis (also known as opinion mining) refers to the process of identifying emotions behind any given written work. With the recent increase in the amount of online content, this technology has gained significant importance. It is a powerful tool that can reveal crucial information about customer feedback, market trends, and public opinion. This could help politicians assess how they are viewed by the public through internet tweets or news headlines, as well as help businesses increase/maintain their customer satisfaction.

However, it is important to note that people can express emotions differently. This can be shown through their writing styles, and their use of sarcasm or irony (Wankhade et al., 2022). In today's world, emojis play an important role in expressing feedback but they can also pose a challenge as they can easily be identified as special characters. Detecting sentiments in different languages can also be quite difficult as their meaning consistently changes due to the context and domain in which they are employed (Wankhade et al., 2022).

Despite facing these challenges, different systems can perform sentiment analysis. These include rule-based, automatic, and hybrid approaches. Rule-based approaches determine the semantic polarity of a given input based on a predefined set of words. They analyse the text, matching each word against a lexicon (i.e. list of words). If found, it classifies it into one of the semantic orientations (positive, negative, and neutral). Automatic approaches use machine learning techniques to address sentiment analysis, while hybrid approaches use a combination of both strategies.

To identify the most effective approach to pre-processing, three different feature sets were created, each with different styles and approaches. This included variations in the number of n-grams and the implementations of different feature normalization techniques such as Trivial Normalisation, TF-IDF (term frequency-document frequency), and PPMI (positive pointwise mutual information).

To help with further test analysis, algorithms such as Lemmatization and Stemming were employed. Lemmatisation is a technique that relies on a dictionary to find the base form of a word. For example, lemmatizing the word "singing" would give you the base form of "sung". Stemming is a more aggressive approach that maps various related words to their common core. For example, the words "expect", "expectation", and "expects" are all reduced to their common core which is "expect". Different feature sets yielded different results.

Related Work

Sentiment Analysis is an ongoing field of research in the text mining field (Wankhade et al., 2022). There are various approaches that have been developed to interpret sentiments, each contributing in diverse ways to expand the particular field.

(Aung & Myo, 2017) explored the importance of analysing students' feedback using a lexicon-based approach to measure the quality of teaching. In this study, they used a database of English sentiment words as a lexical score to determine the polarity of the feedback. They classified the text into one of these semantic orientations: strongly positive, moderately positive, weakly positive, strongly negative, moderately negative, weakly negative, or neutral.

Another existing method explored the prospect of working with machine learning techniques to perform sentiment analysis on text messages. (Bhagat et al., 2020) considered a range of sources, including general tweets and movie reviews to assess the polarity. The research utilised classification algorithms such as SVMs, Naive Bayes, and decision trees. By employing various evaluation metrics, the study identified decision trees and SVMs with the highest accuracy.

(Appel et al.,2016) introduces a novel hybrid approach to tackle the sentiment analysis challenge at the sentence level. This study incorporated Natural Language Processing (NLP), a sentiment lexicon enhanced with the assistance of SentiWordNet, and fuzzy sets to assess the semantic orientation polarity. The outcomes of this study were compared to those attained through the use of Maximum Entropy and Naive Bayes techniques. The results demonstrated through this study showed that the hybrid strategy is not only both more accurate and exact than the Naive Bayes and Maximum Entropy procedures but also the state-of-the-art methods when applied to datasets that contain snippets.

Sentiment analysis is needed in several real-world applications for in-depth research such as the business sector which has used sentiment analysis for its improvement (Wankhade et al., 2022). It has previously found application across various domains, encompassing areas from hotels and airlines to healthcare and the stock market (Zvarevashe and Olugbara 2018). As a result, there are several methods available to address sentiment analysis, showcasing diverse approaches to understanding and interpreting emotional expressions in text.

Experiments and Results

Feature Selection

Throughout this project, I explored three sets for feature generation and selection to determine the most effective approach. Before applying these operations to the reviews, all the words were tokenized (i.e. broken down into individual words). This is done by looping through each document and using the in-built `word_tokenize` function.

I. Combination 1

The first feature set involved a combination of lowercasing all the words, eliminating stopwords (i.e. a set of commonly used words such as and, of, etc.), employing lemmatization, and trivial normalization (Figure 1). For this set, I decided to take an n-gram of 3, which is also known as trigrams.

```
[6] #Combination 1
    def process_one(data_list):
      stoplist = set(stopwords.words('english'))
      lemmatizer = WordNetLemmatizer()
      tokenized_list = []
      for content in data_list:
        #list comprehension
        word_list = [lemmatizer.lemmatize(word) for word in word_tokenize(content.lower()
                     if not word in stoplist]
        tokenized_list.append(word_list)
      return tokenized_list

    training_feature_selection_one = process_one(X_train)
    dev_feature_selection_one = process_one(X_dev)
    test_feature_selection_one = process_one(X_test)
```

Figure 1: Feature set one

For the above code, I imported the external libraries used for `stoplist` and the `WordNetLemmatizer()`.

Trivial normalisation is the count of each term in the document divided by the length of that particular document. To perform the trivial normalisation, I created a loop to run through each document, where it calculates the frequency of each n-gram using the `Counter()` class. It then creates a new dictionary, stored in the variable `tf_dict` where the keys are n-grams and the values are the term frequencies of those n-grams. The `tf_dict` is then appended to a list, stored in `trivial_list`, which is returned, as shown in Figure 2.

```
[10] # Trivial Normalisation
     def trivial_normalisation(ngrams_list):
         trivial_list = []
         for doc in ngrams_list:
             frequencies = Counter(doc)
             tf_dict = {gram: frequencies[gram]/ len(doc) for gram in doc}
             trivial_list.append(tf_dict)
         return trivial_list

     train_values_one = trivial_normalisation(train_ngrams_one)
     dev_values_one = trivial_normalisation(dev_ngrams_one)
     test_values_one = trivial_normalisation(test_ngrams_one)

     train_values_two = trivial_normalisation(train_ngrams_one)
     dev_values_two = trivial_normalisation(dev_ngrams_two)
     test_values_three = trivial_normalisation(test_ngrams_three)

     train_values_one = trivial_normalisation(train_ngrams_one)
     dev_values_two = trivial_normalisation(dev_ngrams_two)
     test_values_three = trivial_normalisation(test_ngrams_three)
```

Figure 2: Trivial Normalisation

II. Combination 2

The second approach employed an approach of converting the words to lowercase, removal of punctuation, eliminating stopwords, employing stemming, and TF-IDF (term frequency-inverse document frequency). This is shown in Figure 3. For this particular combination, I opted for an n-gram of 1, also known as unigrams.

```
#Combination two
def process_two(data_list):
    stoplist = set(stopwords.words('english'))
    st = PorterStemmer()
    tokenized_list = []
    for content in data_list:
        word_list = [st.stem(re.sub(r'[^a-zA-Z0-9]', '', word)) for word in word_tokenize(content.lower()) if not word in stoplist and not re.match(r'[^a-zA-Z0-9]+', word)]
        tokenized_list.append(word_list)

    return tokenized_list

training_feature_selection_two = process_two(X_train)
dev_feature_selection_two = process_two(X_dev)
test_feature_selection_two = process_two(X_test)
```

Figure 3: Feature set two

In the code snippet, I imported the external libraries used for `stoplist` and the `PorterStemmer()`. To remove the punctuation, the `re.sub` function is used to replace any non-alphanumeric character with an empty string, and the `re.match` function is used to filter out words that start with non-alphanumeric characters. I did not use the inbuilt `string.punctuation` function as it was allowing words with punctuation to pass through causing a decrease in the overall accuracy.

TF-IDF (term frequency-inverse document frequency) evaluates how important a word is to a document among a collection of documents (corpus). To perform TF-IDF, as shown in Figure 4, the `calculate_all_tf_idfs` function passes a list of n-grams Within this function, is a nested function, `calculate_tf_idf`, which computes the term frequency-inverse document frequency for a given n-gram. It does this by first calculating the term frequencies, which is how often a term occurs in a document, using the `Counter()` class, and then calculating the IDF which considers how rare/common an n-gram is across all documents. The function returns the product for term frequency(TF) and inverse document frequency (IDF).

```
[11] def calculate_all_tfidfs(ngrams_list):
         def calculate_tfidf(gram, doc, doc_frequencies, num_docs):
             frequencies = Counter(doc)
             tf = frequencies[gram] / len(doc)
             idf = np.log(num_docs / (doc_frequencies[gram] + 1))
             return tf * idf

         num_docs = len(ngrams_list)
         doc_frequencies = Counter(gram for doc in ngrams_list for gram in set(doc))

         allTfIDfs = []

         for doc in ngrams_list:
             tfIdfs = {gram: calculate_tfidf(gram, doc, doc_frequencies, num_docs) for gram in doc}
             allTfIDfs.append(tfIdfs)

         return allTfIDfs

     #Process one
     train_tfidf_values_one = calculate_all_tfidfs(train_ngrams_one)
     dev_tfidf_values_one = calculate_all_tfidfs(dev_ngrams_one)
     test_tfidf_values_one = calculate_all_tfidfs(test_ngrams_one)

     #Process two
     train_tfidf_values_two = calculate_all_tfidfs(train_ngrams_two)
     dev_tfidf_values_two = calculate_all_tfidfs(dev_ngrams_two)
     test_tfidf_values_two = calculate_all_tfidfs(test_ngrams_two)

     #Process three
     train_tfidf_values_three = calculate_all_tfidfs(train_ngrams_three)
     dev_tfidf_values_three = calculate_all_tfidfs(dev_ngrams_three)
     test_tfidf_values_three = calculate_all_tfidfs(test_ngrams_three)
```

Figure 4: TF-IDF implementation

The main loop in `calculate_all_tf_idfs` iterates over each document, calculating the TF-IDF scores using the nested function, and stores the result in the variable `allTfIDfs`. The result obtained is a dictionary where the keys are the n-grams and the values are the TF-IDF scores.

III. Combination 3

My third and final approach used a combination of removal of punctuation. I applied stemming and PPMI (positive pointwise mutual information), as shown in Figure 5. For this combination, I used an n-gram of 2, also known as bigrams.

```
[8] def process_three(data_list):
        tokenized_list = []
        st = PorterStemmer()
        for content in data_list:
            word_list = [st.stem(re.sub(r'[^a-zA-Z0-9]', '', word)) for word in word_tokenize(content.lower()) if not re.search(r'[^a-zA-Z0-9]+', word)]
            tokenized_list.append(word_list)
        return tokenized_list

    training_feature_selection_three = process_three(X_train)
    dev_feature_selection_three = process_three(X_dev)
    test_feature_selection_three = process_three(X_test)
```

Figure 5: Feature set three

PPMI is often used to capture the semantic relationship between words. It measures maximising the probability of two words occurring together divided by the probability of each word. The use of the logarithm ensures that we focus on the positive associations and max(0,) function helps eliminate negative values.

Unfortunately, I didn't have time to code PPMI but if I did implement it, I would count the occurrences of each n-gram and then construct a co-occurrence matrix to store the counts between words. Using the counts I would calculate the PPMI frequency normalisation technique for each n-gram.

I selected these three particular pre-processing steps, to observe the individual effects that stopwords, punctuation, and lowercase had on the dataset. While I couldn't complete the computation of PPMI, my experimentation with the two different feature sets revealed that combination two received the highest accuracy. As I found TF-IDF to be the best frequency normalisation technique, I used this metric throughout the project.

Feature Generation

The parameters `data_list` is a list of all the tokenized documents along with the respective pre-processing techniques applied. Looping through each document, `calculate_ngram` uses an inbuilt `ngrams` function to generate the respective ngrams for each word in the document. The `grams_string` converts the ngrams into the string and returns a resulting string of ngrams.

```python
def calculate_ngrams(data_list):
    ngrams_list = []
    for content in data_list:
        n_grams = list(ngrams(content, 1))
        grams_string = [' '.join(gram) for gram in n_grams]
        ngrams_list.append(grams_string)
    return ngrams_list

#Process 1
train_ngrams_one = calculate_ngrams(training_feature_selection_one)
dev_ngrams_one = calculate_ngrams(dev_feature_selection_one)
test_ngrams_one = calculate_ngrams(test_feature_selection_one)

#Process 2
train_ngrams_two = calculate_ngrams(training_feature_selection_two)
dev_ngrams_two = calculate_ngrams(dev_feature_selection_two)
test_ngrams_two = calculate_ngrams(test_feature_selection_two)

#Process 3
train_ngrams_three = calculate_ngrams(training_feature_selection_three)
dev_ngrams_three = calculate_ngrams(dev_feature_selection_three)
test_ngrams_three = calculate_ngrams(test_feature_selection_three)
```

Figure 6: N-gram generation

Choosing a particular n-gram depended upon the accuracy it would receive after passing it through the Naive Bayes. Originally, I had different ngrams for each feature set however through several trials, it was shown that by using an n-gram of one, the accuracy would consistently be the highest and by using an n-gram of three, the accuracy would be the lowest.

Data Splits

Before applying the pre-processing steps, I split the data using the in-built `train_test_spilt` function from skict-learn. I combined the positive and negative reviews by concatenating the two lists in the variable `TotalDataset`. It is important to have a balanced representation of different classes in the training data otherwise it could lead to biased predictions. Each element of `TotalDataset` is a tuple containing a review (data) and a label (positive or negative sentiment).

```
[5] from sklearn.model_selection import train_test_split

    TotalDataset = positive_reviews + negative_reviews

    data = [review[0] for review in TotalDataset]
    labels = [label[1] for label in TotalDataset]

    #Spillting the data into 80-10-10
    X_train, X_temp, y_train, y_temp = train_test_split(data, labels, test_size=0.2, random_state=42)
    X_dev, X_test, y_dev, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

Figure 7: Data Splits

Using the `train_test_spilt` function, the data was divided into an 80-10-10 split, as shown in Figure 1. This is done to ensure there is a balance between the training set (80%) to ensure the model generalizes well to unseen data, a development set (10%) for fine-tuning and preventing overfitting, and the testing set (10%) for unbiased evaluation.

The way the function works is by splitting the entire dataset into a training set `(X_train,y_train)` and a temporary set `(X_temp,y_temp)` with an 80-40 split. This temporary split is further split into the development set and the testing set, preserving the class distribution observed in the original dataset. The `random_state` variable controls the amount of shuffling applied to the data before the split.

Naive Bayes

Naive Bayes is a classification algorithm based on Bayes' theorem. This model is referred to as naive as it assumes that all the features in the data are independent of each other. Using the above three feature sets, I ran them through both my custom implementation and the built-in function.

I. Implementation from Scratch

My implementation involved creating separate functions for calculating the prior, likelihood, and prediction. The `calculate_prior` function calculates the prior probabilities of each class based on the class labels in the training set (`y_train`).

It counts the occurrences of each class and divides it by the total number of samples to obtain the probability of each class.

```python
import numpy as np
from sklearn.metrics import accuracy_score, classification_report
from collections import defaultdict

def calculate_prior(y_train):
    class_counts = defaultdict(int)
    for label in y_train:
        class_counts[label] += 1
    total_samples = len(y_train)
    class_probabilities = {label: count / total_samples for label, count in class_counts.items()}
    return class_probabilities
```

Figure 8: Prior Probability

Likelihood refers to the probability of observing a particular feature given the class variable in this case, 'positive' or 'negative'. The `calculate_likelihood` iterates through the training data, counting the occurrence of each feature for their respective classes and normalizes it by the total number of samples in that class. This provided the conditional probabilities of each feature.

```python
def calculate_likelihood(tfidf_matrix, y_train):
    class_counts = defaultdict(int)
    feature_counts = defaultdict(lambda: defaultdict(float))

    for i, label in enumerate(y_train):
        class_counts[label] += 1
        for j, value in enumerate(tfidf_matrix[i]):
            feature_counts[label][j] += value

    likelihoods = defaultdict(dict)
    for label in class_counts:
        total_samples_in_class = class_counts[label]
        likelihoods[label] = {feature: count / total_samples_in_class for feature, count in feature_counts[label].items()}

    return likelihoods

def predict(tfidf_matrix, prior_probabilities, likelihoods):
    predictions = []

    # Precompute normalized likelihoods for each label
    normalized_likelihoods = {label: {feature: count / sum(likelihoods[label].values()) for feature, count in likelihoods[label].items()} for label in prior_probabilities}

    for sample in tfidf_matrix:
        max_prob = float('-inf')
        predicted_label = None

        for label in prior_probabilities:
            log_prob = sum([sample[feature] * normalized_likelihoods[label].get(feature, 0) for feature in range(len(sample))])
            log_prob += prior_probabilities[label]

            if log_prob > max_prob:
                max_prob = log_prob
                predicted_label = label

        predictions.append(predicted_label)

    return predictions

def calculate_accuracy(y_true, y_pred):
    return accuracy_score(y_true, y_pred)
```

Figure 9: Naive Bayes Implementation

Last but not least, the `predict` function predicts the class label for each sample by computing the log probability for each class and choosing the class that receives the highest accuracy. This evaluation is performed on the development set.

However, my code is giving me identical accuracies for the three different feature sets. I attempted various solutions, including introducing a scaling factor, but none of them have made a difference.

```
Accuracy for Combination one: 0.4775

Classification Report for Combination one
:              precision    recall  f1-score   support

    negative       0.48      1.00      0.65       191
    positive       0.00      0.00      0.00       209

    accuracy                           0.48       400
   macro avg       0.24      0.50      0.32       400
weighted avg       0.23      0.48      0.31       400
```

Figure 10: Accuracy for feature set one

I delved into debugging to compare the prior probabilities which remain the same for each feature set as they're derived from the training labels and although the likelihoods are different, the problem persists.

```
Accuracy for Combination two: 0.4775

Classification Report for Combination two
:              precision    recall  f1-score   support

    negative       0.48      1.00      0.65       191
    positive       0.00      0.00      0.00       209

    accuracy                           0.48       400
   macro avg       0.24      0.50      0.32       400
weighted avg       0.23      0.48      0.31       400
```

Figure 11: Accuracy for feature set one

```
Accuracy for Combination three: 0.4775

Classification Report for Combination three
:              precision    recall  f1-score   support

    negative       0.48      1.00      0.65       191
    positive       0.00      0.00      0.00       209

    accuracy                           0.48       400
   macro avg       0.24      0.50      0.32       400
weighted avg       0.23      0.48      0.31       400
```

Figure 12: Accuracy for feature set three

II. Implementation from built-in class

Using the `MultinomialNB()` on the three different feature sets, I fit the model using the training matrix and the corresponding labels. Upon evaluation of the development matrix, it became evident that combination two, which involves removing stopwords, punctuation, lowercasing transformation, stemming, and utilizing TF-IDF achieved the highest accuracy of 83%. On the other hand, combination three gave me the lowest accuracy of 72% indicating the importance of stopword removal during the pre-processing stage.

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

# Create a Multinomial Naive Bayes classifier
nb_classifier = MultinomialNB()

# Train the classifier
nb_classifier.fit(train_tfidf_matrix_one, y_train)
#Evaluating on the development matrix
y_pred_one = nb_classifier.predict(dev_tfidf_matrix_one)
accuracy_one = accuracy_score(y_dev, y_pred_one)
print(f"Combination one Accuracy: {accuracy_one}")
ClassificationReport_one = classification_report(y_dev, y_pred_one)
print(f"Combination one Classification Report: {ClassificationReport_one}")

#Process two
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

nb_classifier = MultinomialNB()

nb_classifier.fit(train_tfidf_matrix_two, y_train)
y_pred_two = nb_classifier.predict(dev_tfidf_matrix_two)

accuracy_two = accuracy_score(y_dev, y_pred_two)
print(f"Combination two Accuracy: {accuracy_two}")
ClassificationReport_two = classification_report(y_dev, y_pred_two)
print(f"Combination two Classification Report: {ClassificationReport_two}")

#Process three
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

nb_classifier = MultinomialNB()

nb_classifier.fit(train_tfidf_matrix_three, y_train)
y_pred_three = nb_classifier.predict(dev_tfidf_matrix_three)

accuracy_three = accuracy_score(y_dev, y_pred_three)
ClassificationReport_three = classification_report(y_dev, y_pred_three)
print(f"Combination three Accuracy: {accuracy_three}")
ClassificationReport_three = classification_report(y_dev, y_pred_three)
print(f"Combination three Classification Report: {ClassificationReport_three}")
```

Figure 13: MultinomialNB() implementation

```
Combination one Accuracy: 0.8125

Combination one Classification Report
:              precision    recall  f1-score   support

    negative       0.75      0.92      0.82       191
    positive       0.91      0.71      0.80       209

    accuracy                           0.81       400
   macro avg       0.83      0.82      0.81       400
weighted avg       0.83      0.81      0.81       400

Combination two Accuracy: 0.8375

Combination two Classification Report
:              precision    recall  f1-score   support

    negative       0.79      0.90      0.84       191
    positive       0.90      0.78      0.83       209

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

Combination three Accuracy: 0.7725

Combination three Classification Report
:              precision    recall  f1-score   support

    negative       0.69      0.95      0.80       191
    positive       0.93      0.61      0.74       209

    accuracy                           0.77       400
   macro avg       0.81      0.78      0.77       400
weighted avg       0.81      0.77      0.77       400
```

Figure 14: Evaluation Metrics

SGD-based classification and SVMs

Stochastic Gradient Descent is an optimisation algorithm. This algorithm is applied in many machine learning applications due to its speed, efficiency, and stability. I evaluated each of my three features on the Logistic Regression and Support Vector Machines (SVM) packages to determine the best of the three methods.

Logistic Regression (LR) is a classification algorithm used for binary classification problems (two classes) while SVM is a powerful supervised learning machine algorithm used for both classification and regression tasks.

In the code snippets, as shown below in Figures 15 and 17, I used the `sckit_learn` library to perform logistic regression, and SVM's. I assessed the accuracy across the three different combinations of training data for both Logistic Regression and SVM. These sets were represented as a matrix of TF-IDF values. The process involves training the model for each combination, followed by the evaluation of the development set.

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

#Combination one
model = LogisticRegression()
model.fit(train_tfidf_matrix_one, y_train)

# Evaluate on the development set
dev_predictions_one = model.predict(dev_tfidf_matrix_one)
logistic_regresion_accuracy_one = accuracy_score(y_dev, dev_predictions_one)
print(f"Logistic Regression Accuracy for Combination one: {logistic_regresion_accuracy_one}")

ClassificationReport_one = classification_report(y_dev, dev_predictions_one)
print(f"\nCombination one Classification Report\n: {ClassificationReport_one}")

#Combination two
model = LogisticRegression()
model.fit(train_tfidf_matrix_two, y_train)

# Evaluate on the development set
dev_predictions_two = model.predict(dev_tfidf_matrix_two)
logistic_regresion_accuracy_two = accuracy_score(y_dev, dev_predictions_two)
print(f"Logistic Regression Accuracy for Combination two: {logistic_regresion_accuracy_two}")

ClassificationReport_two = classification_report(y_dev, dev_predictions_two)
print(f"\nCombination one Classification Report\n: {ClassificationReport_two}")

#Combination three
model = LogisticRegression()
model.fit(train_tfidf_matrix_three, y_train)

# Evaluate on the development set
dev_predictions_three = model.predict(dev_tfidf_matrix_three)
logistic_regresion_accuracy_three = accuracy_score(y_dev, dev_predictions_three)
print(f"Logistic Regression Accuracy for Combination three: {logistic_regresion_accuracy_three}")

ClassificationReport_three = classification_report(y_dev, dev_predictions_three)
print(f"\nCombination one Classification Report\n: {ClassificationReport_three}")
```

Figure 15: Logistic Regression Implementation

```
Logistic Regression Accuracy for Combination one: 0.85

Combination one Classification Report
:               precision    recall  f1-score   support

    negative       0.82      0.87      0.85       191
    positive       0.88      0.83      0.85       209

    accuracy                           0.85       400
   macro avg       0.85      0.85      0.85       400
weighted avg       0.85      0.85      0.85       400

Logistic Regression Accuracy for Combination two: 0.8575

Combination one Classification Report
:               precision    recall  f1-score   support

    negative       0.86      0.84      0.85       191
    positive       0.86      0.88      0.87       209

    accuracy                           0.86       400
   macro avg       0.86      0.86      0.86       400
weighted avg       0.86      0.86      0.86       400

Logistic Regression Accuracy for Combination three: 0.805

Combination one Classification Report
:               precision    recall  f1-score   support

    negative       0.76      0.87      0.81       191
    positive       0.87      0.74      0.80       209

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

Figure 16: Evaluation metrics

```python
from sklearn.svm import SVC

#Combination one
model = SVC()
model.fit(train_tfidf_matrix_one, y_train)
# Evaluate on the development set
dev_predictions_one = model.predict(dev_tfidf_matrix_one)
SVM_accuracy_one = accuracy_score(y_dev, dev_predictions_one)
print(f"SVM Accuracy for Combination one: {SVM_accuracy_one}")

ClassificationReport_one = classification_report(y_dev, dev_predictions_one)
print(f"\nCombination one Classification Report\n: {ClassificationReport_one}")

#Combination two
model = SVC()
model.fit(train_tfidf_matrix_two, y_train)
dev_predictions_two = model.predict(dev_tfidf_matrix_two)
SVM_accuracy_two = accuracy_score(y_dev, dev_predictions_two)
print(f"SVM Accuracy for Combination two: {SVM_accuracy_two}")

ClassificationReport_two = classification_report(y_dev, dev_predictions_two)
print(f"\nCombination two Classification Report\n: {ClassificationReport_two}")

#Combination three
model = SVC()
model.fit(train_tfidf_matrix_three, y_train)
dev_predictions_three = model.predict(dev_tfidf_matrix_three)
SVM_accuracy_three = accuracy_score(y_dev, dev_predictions_three)
print(f"SVM Accuracy for Combination three: {SVM_accuracy_three}")

ClassificationReport_three = classification_report(y_dev, dev_predictions_three)
print(f"\nCombination three Classification Report\n: {ClassificationReport_three}")
```

Figure 17: SVM Implementation

```
SVM Accuracy for Combination one: 0.865

Combination one Classification Report
:               precision    recall  f1-score   support

    negative       0.88      0.83      0.85       191
    positive       0.85      0.90      0.87       209

    accuracy                           0.86       400
   macro avg       0.87      0.86      0.86       400
weighted avg       0.87      0.86      0.86       400

SVM Accuracy for Combination two: 0.87

Combination two Classification Report
:               precision    recall  f1-score   support

    negative       0.91      0.81      0.86       191
    positive       0.84      0.93      0.88       209

    accuracy                           0.87       400
   macro avg       0.88      0.87      0.87       400
weighted avg       0.87      0.87      0.87       400

SVM Accuracy for Combination three: 0.8575

Combination three Classification Report
:               precision    recall  f1-score   support

    negative       0.89      0.81      0.84       191
    positive       0.84      0.90      0.87       209

    accuracy                           0.86       400
   macro avg       0.86      0.86      0.86       400
weighted avg       0.86      0.86      0.86       400
```

Figure 18: Evaluation Metrics for SVM

I. Hyparameter Optimisation for LR and SVM

The function `logistic_regression_hyperparameter` and `svm_hyperparameter` perform the hyperparameter tuning for logistic regression and SVM models respectively using a random search approach. The functions take in the training, development, and test matrices along with their corresponding labels. These matrices are selected based on their performance from the `sckit_learn` library. Specifically, the best matrix is chosen for having the highest accuracy.

In Logistic Regression, the parameters I chose to tune included regularisation strength (`C`), penalty type (`penalty`), solver type (`solver`), and maximum iterations (`max_iter`) while in SVM's, the parameters that were tuned included regularisation strength parameters(`C`), kernel type (`kernel`), and gamma value (`gamma`).

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
import random
from sklearn.svm import SVC

def logistic_regression_hyperparameter(train_matrix, dev_matrix, test_matrix, y_train, y_dev, y_test):
    # Define hyperparameter grid
    hyperparameters = {
        'C': [0.01, 0.1, 1, 10, 100],
        'penalty': ['l2'],
        'solver': ['lbfgs'],
        'max_iter': [100, 200, 300]  # Add a range of max_iter values
    }

    best_accuracy = 0
    best_hyperparameters = None

    # Try 5 combinations
    for _ in range(5):
        # Randomly select hyperparameters
        current_hyperparameters = {
            'C': random.choice(hyperparameters['C']),
            'penalty': random.choice(hyperparameters['penalty']),
            'solver': random.choice(hyperparameters['solver']),
            'max_iter': random.choice(hyperparameters['max_iter'])
        }

        # Create and train logistic regression model
        model = LogisticRegression(**current_hyperparameters)
        model.fit(train_matrix, y_train)

        # Evaluate on dev matrix
        y_dev_pred = model.predict(dev_matrix)
        dev_accuracy = accuracy_score(y_dev, y_dev_pred)

        # Check if current combination is the best
        if dev_accuracy > best_accuracy:
            best_accuracy = dev_accuracy
            best_hyperparameters = current_hyperparameters

    # Apply best hyperparameters on test set
    best_model = LogisticRegression(**best_hyperparameters)
    best_model.fit(train_matrix, y_train)
    y_test_pred = best_model.predict(test_matrix)
    test_accuracy = accuracy_score(y_test, y_test_pred)
    ClassificationReport = classification_report(y_test, y_test_pred)

    print("Dev Set Accuracy (Best):", best_accuracy)
    print("Test Set Accuracy:", test_accuracy)
    print("\nClassification report\n", ClassificationReport)

    return best_hyperparameters, best_accuracy, test_accuracy

logistic_regression_hyperparameter(train_tfidf_matrix_two, dev_tfidf_matrix_two, test_tfidf_matrix_two, y_train, y_dev, y_test)
```

Figure 19: Hyperparameter Optimisation for Logistic Regression

```
Dev Set Accuracy (Best): 0.87
Test Set Accuracy: 0.8675

Classification report
              precision    recall  f1-score   support

    negative       0.84      0.88      0.86       187
    positive       0.89      0.85      0.87       213

    accuracy                           0.87       400
   macro avg       0.87      0.87      0.87       400
weighted avg       0.87      0.87      0.87       400

({'C': 10, 'penalty': 'l2', 'solver': 'lbfgs', 'max_iter': 300}, 0.87, 0.8675)
```

Figure 20: Evaluation Metrics + Best Hyperparameters for LR

```python
def svm_hyperparameter(train_matrix, dev_matrix, test_matrix, y_train, y_dev, y_test):
    # Define hyperp__arameter grid
    hyperparameters = {
        'C': [0.1, 1, 10, 100],
        'kernel': ['linear', 'rbf'],
        'gamma': ['scale', 'auto']
    }

    best_accuracy = 0
    best_hyperparameters = None

    # Try 5 combinations
    for i in range(5):
        # Randomly select hyperparameters
        current_hyperparameters = {
            'C': random.choice(hyperparameters['C']),
            'kernel': random.choice(hyperparameters['kernel']),
            'gamma': random.choice(hyperparameters['gamma'])
        }

        # Create and train SVM model
        model = SVC(**current_hyperparameters)
        model.fit(train_matrix, y_train)

        # Evaluate on dev matrix
        y_dev_pred = model.predict(dev_matrix)
        dev_accuracy = accuracy_score(y_dev, y_dev_pred)

        # Check if current combination is the best
        if dev_accuracy > best_accuracy:
            best_accuracy = dev_accuracy
            best_hyperparameters = current_hyperparameters

    # Apply best hyperparameters on test set
    best_model = SVC(**best_hyperparameters)
    best_model.fit(train_matrix, y_train)
    y_test_pred = best_model.predict(test_matrix)
    test_accuracy = accuracy_score(y_test, y_test_pred)
    ClassificationReport = classification_report(y_test, y_test_pred)

    print("\nBest Hyperparameters:", best_hyperparameters)
    print("Dev Set Accuracy (Best):", best_accuracy)
    print("Test Set Accuracy:", test_accuracy)
    print("\nClassification report\n", ClassificationReport)

    return best_hyperparameters, best_accuracy, test_accuracy

svm_hyperparameter(train_tfidf_matrix_two, dev_tfidf_matrix_two, test_tfidf_matrix_two, y_train, y_dev, y_test)
```

Figure 21: Hyperparameter Optimisation for SVM

```
Best Hyperparameters: {'C': 1, 'kernel': 'rbf', 'gamma': 'scale'}
Dev Set Accuracy (Best): 0.87
Test Set Accuracy: 0.8625

Classification report
              precision    recall  f1-score   support

    negative       0.83      0.88      0.86       187
    positive       0.89      0.85      0.87       213

    accuracy                           0.86       400
   macro avg       0.86      0.86      0.86       400
weighted avg       0.86      0.86      0.86       400

({'C': 1, 'kernel': 'rbf', 'gamma': 'scale'}, 0.87, 0.8625)
```

Figure 22: *Evaluation Metrics + Best Hyperparameters for LR*

The functions iterate through the five random combinations of hyperparameters, using the `random.choice` function. They train the models with these configurations, evaluate them on the development set, and keep track of the best-performing set of hyperparameters. After the random search, the function trains the models with the respective best hyperparameters and then evaluates their performance on the test set.

BERT

Bert stands for Bidirectional Encoder Representations from Transformers is a natural language processing (NLP) pre-training technique. It can understand that the meaning of a word can change based on the words around it in a sentence, allowing for more accurate interpretations.

Unfortunately, I didn't have enough time to complete the implementation of BERT. However, I did explore the code and tried to implement it as shown below in Figure 18. The code was designed for fine-tuning a `DistilBERT` model using the Hugging Face transformers library.

```python
#BERT
import os
from google.colab import drive
drive.mount('/content/drive')

from transformers import DistilBertForSequenceClassification, Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir='/content/drive/MyDrive/bert',          # output directory
    num_train_epochs=3,              # total number of training epochs
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=64,   # batch size for evaluation
    warmup_steps=500,                # number of warmup steps for learning rate scheduler
    weight_decay=0.01,               # strength of weight decay
    logging_dir='/content/drive/MyDrive/bertlogs',          # directory for storing logs
    logging_steps=10,
)

model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased")

trainer = Trainer(
    model=model,                         # the instantiated Transformers model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=X_train,               # training dataset
    eval_dataset=X_dev                   # evaluation dataset
)

trainer.train()
model.save()
model.eval()
```

Figure 23: BERT implementation

I started by mounting Google Drive and importing the necessary libraries. The `trainer` handles the training process. However, this code wasn't running as I consistently got an error regarding the import statements.

Discussion

Model	Feature Set	Test set
Naive Bayes (My implementation)	Combination one	Accuracy: 0.4655 Precision: 0.23 Recall: 0.50 F1-Score: 0.47
	Combination two	Accuracy: 0.4675 Precision: 0.24 Recall: 0.58 F1-Score: 0.48
	Combination three	Accuracy: 0.4675 Precision: 0.23 Recall: 0.50 F1-Score: 0.47
Naive Bayes (scikit-learn)	Combination one	Accuracy: 0.8375 Precision: 0.85 Recall: 0.84 F1-Score: 0.84
	Combination two	Accuracy: 0.865 Precision: 0.87 Recall: 0.87 F1-Score: 0.86
	Combination three	Accuracy: 0.865 Precision: 0.81 Recall: 0.78 F1-Score: 0.77
Logistic Regression	Combination one	Accuracy: 0.855 Precision: 0.86 Recall: 0.86 F1-Score: 0.85

	Combination two	Accuracy: 0.875 Precision: 0.87 Recall: 0.88 F1-Score: 0.88
	Combination three	Accuracy: 0.8425 Precision: 0.85 Recall: 0.85 F1-Score: 0.84
SVM	Combination one	Accuracy: 0.86 Precision: 0.86 Recall: 0.86 F1-Score: 0.86
	Combination two	Accuracy: 0.8625 Precision: 0.86 Recall: 0.86 F1-Score: 0.86
	Combination three	Accuracy: 0.8675 Precision: 0.87 Recall: 0.87 F1-Score: 0.87
Hyparamter Optimisation for Logistic Regression	Best Feature Set [Combination two]	Accuracy: 0.8675 Precision: 0.87 Recall: 0.87 F1-Score: 0.87
Hyparamter Optimisation for SVM	Best Feature Set [Combination two]	Accuracy: 0.8625 Precision: 0.86 Recall: 0.86 F1-Score: 0.86

Choosing the best model depends on the specific characteristics of your data or the complexity of the problem. Several papers compared the differences between the different methods. (Rahat et al., 2019) stated that SVMs are one of the most powerful algorithms for text classification and have proved to work better than Naive Bayes in Sentimental Analysis. SVMs can also perform well due to their ability to handle non-linear relationships (i.e. when the decision boundary can't separate the two classes). However, the disadvantage of using SVMs is that they can be highly computationally expensive and can take a long time to run especially for large datasets.

Logistic Regression has also proven to work better than Naive Bayes due to its ability to model complex relationships between the feature and target variable as well as capture linear and non-linear relationships.

Naive Bayes(my implementation) could have the disadvantage of assuming the features are independent of each other which could lead to lower performance especially if features are correlated. Using sckit-learn's implementation is highly beneficial as it's easy to implement and can be adaptable to different data types. It has demonstrated a strong performance on the test data, indicating its ability to generalise well to unseen data.

Tuning of Hyperparameters is extremely important in the performance of the model. Upon tuning the hyperparameters for both SVM and Logistic regression, the accuracy improved or remained unchanged. There was a slight increase in tuning the hyperparameters for logistic regression during the evaluation of the development set. However, for SVMs it remained unchanged.

Before Hyperparameter optimization, SVMs showed the highest accuracy and were considered the best-performing mode. However, following parameter tuning for both of the models, Logistic Regression became the top-performing model.

Despite not having complete BERT, BERT has proven it can achieve state-of-the-art results (Cutkosky et al., 2020). It can make more accurate interpretations due to its ability to consider the surrounding words, can generalise well with unlabelled data, and also has the ability to understand that words have different meanings based on the context/domain they're employed in.

Conclusions and Future Work

In conclusion the best-performing model depends upon the specific demands of the task and the complexity of the problem. Various pre-processing techniques highlighted the importance of tasks such as eliminating stopwords and punctuation. This report showed that Logistic Regression is the best-performing model after hyperparameter tuning.

To further improve the test results, given BERT's success rate, I would incorporate a fine-tuned model to demonstrate the way it contributes to sentimental analysis as well as try additional feature selection methods. Considering SVM's high performance, before hyparameter tuning, I would try an optimise their computational efficiency to allow for a faster convergence rate.

References

Appel, O., Chiclana, F., Carter, J., & Fujita, H. (2016). A hybrid approach to the sentiment analysis problem at the sentence level. *Knowledge-Based Systems*, *108*, 110-124.

Aung, K. Z., & Myo, N. N. (2017, May). Sentiment analysis of students' comment using lexicon based approach. In *2017 IEEE/ACIS 16th international conference on computer and information science (ICIS)* (pp. 149-154). IEEE.

Bhagat, A., Sharma, A., & Chettri, S. (2020). Machine learning based sentiment analysis for text messages. *International Journal of Computing and Technology*.

Cutkosky, A., & Mehta, H. (2020, November). Momentum improves normalized sgd. In *International conference on machine learning* (pp. 2260-2268). PMLR.

Rahat, A. M., Kahir, A., & Masum, A. K. M. (2019, November). Comparison of Naive Bayes and SVM Algorithm based on sentiment analysis using review dataset. In *2019 8th International Conference System Modeling and Advancement in Research Trends (SMART)* (pp. 266-270). IEEE.

Wankhade, M., Rao, A. C. S., & Kulkarni, C. (2022). A survey on sentiment analysis methods, applications, and challenges. *Artificial Intelligence Review*, *55*(7), 5731-5780.

Zvarevashe, K., & Olugbara, O. O. (2018, March). A framework for sentiment analysis with opinion mining of hotel reviews. In *2018 Conference on information communications technology and society (ICTAS)* (pp. 1-4). IEEE.