Part A – Sentiment Analysis

Table of Contents

Abstract

This project aims to develop text classification models using different Machine Learning (ML) and Natural Language Processing (NLP) algorithms to determine the sentiment polarity (positive or negative) of a given movie review from the Internet Movie Database (IMDb). The project includes assessing the performance of a Naïve Bayes, Logistic Regression and Support Vector Machine (SVM) classifier in determining the sentiment polarity, while evaluating finer details such as which feature sets work best for each model. This will involve feature selection by implementing and systematically choosing a combination of n-grams, stop words removal, TF-IDF, lemmatization or stemming, to construct a document-term matrix for input to the models. In addition, the performance of a custom Naïve Bayes implementation will be compared against an existing one from Scikit-learn. Finally, the project entails experimentation with different versions of the large language model, BERT, to explore its effectiveness in performing sentiment classification on the same IMDb dataset the initial models were evaluated on.

By the conclusion of the project, my objective is to have a better understanding of classification, specifically the importance of relevant feature generation, selection and extraction, and appropriate model selection in the context of using NLP for sentiment analysis. Furthermore, I aspire to improve my understanding of the capabilities and limitations of pre-trained large language models which will have an ever-increasing influence in societal applications.

Introduction and Motivation

Importance of sentiment analysis

Sentiment analysis is very important within NLP because it demonstrates a machine's capability to comprehend the underlying meaning or sentiment conveyed in a given input. In terms of professional use cases, sentiment analysis is important especially amidst the abundance of public data, enabling it to enhance efficiency and better-informed decision-making across a wide variety of tasks. For example, it could be used to analyse customer feedback on a given product, or to analyse the social media sentiment towards a particular company, which could then be used to improve marketing strategies.

Difficulties of sentiment analysis

Aside from the inherent challenges in creating a text classification model, the complexity of sentiment analysis increases significantly when figures of speech are encountered, such as irony and sarcasm. Moreover, further nuances in text such as negations, subjective expressions, ambiguous words, and even multi-polarity, make creating an accurate model more challenging. This complexity is not only confined to models but can pose difficulties for humans too. An example of a difficult sentence to classify is: "The movie was… interesting, to say the least."

Different methods available for sentiment analysis

There are multiple ways of conducting sentiment analysis, ranging from lexicon-based systems which identify semantic and syntactic patterns, to the ML approaches explored in this project. While the selection of the method should depend on the task at hand, lexicon-based systems often are not able to cope well with the difficulties previously discussed such as irony due to the stricter rules in place. However, they are a less expensive option (Yilmaz, 2023) and can be more readily accessible with publicly available resources such as SentiWordNet. Lastly, hybrid approaches and ensemble methods offer alternative strategies, combining rule-based and (potentially multiple) ML methods together.

Intended testing

I intend to test and evaluate the effectiveness of different ML sentiment analysis algorithms in the context of movie reviews from IMDb. The sample of 2000 positive and negative movie reviews provided by the Large Movie Review Dataset will allow sufficient data for training, followed by thorough evaluation of the accuracy on the development set. Additionally, I intend to test the effectiveness of different feature extraction techniques and explore whether specific models exhibit better compatibility with particular feature selection methods.

Different approaches taken and how they worked

Throughout the project I experimented with a combination of feature selection methods and models including implementing TF-IDF and Naïve Bayes from scratch. Unfortunately, time constraints hindered further exploration of novel models, however, after training the logistic regression and SVM model, I was intrigued by the prospect of combining them into an ensemble method to produce a more robust classifier. This is discussed in greater detail in the future work section. Furthermore, in the Naïve Bayes implementation, it was observed that adding P (class | word) together for each word in the document, as opposed to multiplication, provided superior results due to numerical instability when dealing with multiplication of many small probabilities.

Related work

In the introductory section, I divided existing sentiment analysis methods into two primary categories: lexicon-based techniques and ML techniques, such Naïve Bayes, Logistic regression and SVM which form the focus of our exploration in this project. Historically, Naïve Bayes and SVM have been the most widely used algorithms in solving the sentiment analysis problem in the ML domain (Medhat et al., 2014). However, well respected studies, including “Comparing and combining sentiment analysis methods" by Gonçalves et al. (2013), which compared 8 different cutting-edge algorithms at the time for sentiment analysis, have concluded that the effectiveness of algorithms vary depending on the nature of the text sources.

Delving into more contemporary research, a recent survey titled “A survey on sentiment analysis methods, applications, and challenges” (Wankhade et al., 2022) suggests that significant contributions to the accuracy of models are rooted in feature generation and selection. This entails considering appropriate n-gram selection, accounting for punctuation, pragmatic features and even the extracting the use of emojis and slang words which can indicate sentiment. Moreover, the techniques to extract the most valuable generated features is as important. Methods include TF-IDF, Part-of-Speech (PoS) tagging, Bag-of-Words (BoW) and handling negations. However, it must be noted that each method has their drawback. For instance, BoW and TF-IDF approaches may lack semantic understanding, while PoS tagging may be constrained by limited knowledge of the context.

Lastly, in recent literature, a noteworthy trend in sentiment analysis research indicates a growing preference in hybrid and ensemble methods. A notable example is found in a study conducted by Chiew et al. (2019), where a hybrid ensemble feature selection framework is introduced, specifically for a phishing detection system. This approach produces effective baseline features, particularly when combined with the Random Forest classifier, remarkably achieving a 94.6% accuracy using only 20.8% of the original features – surpassing the performance of other classifiers like SVM, Naive Bayes, C4.5, JRip, and PART.

Experiments and Results

Feature Generation using n-grams

```python
# Creates a vectorizer to convert preprocessed data to a sparse matrix
# where each row represents a document (review), and each column represents
# a unique n-gram (a single word if unigram)
for i in range(1, 4):
    for j in range(i, 4):
        vectorizer = CountVectorizer(ngram_range=(i, j))
```
…
```python
print(f"N-gram: {i}, {j} accuracy: {dev_accuracy}")
```

N-gram	Accuracy random seed 1	Accuracy random seed 2	Accuracy random seed 3
(1, 1)	0.80	0.82	0.82
(1, 2)	0.83	0.84	0.86
(1, 3)	0.83	0.84	0.87
(2, 2)	0.81	0.84	0.85
(2, 3)	0.82	0.83	0.86
(3, 3)	0.78	0.79	0.79

Table 1: N-gram accuracy across different seeds

The trade-off in selecting the n-gram size to generate features is between precision and recall. Unigrams, for instance, place more emphasis on individual words, while larger n-grams incorporate more context by considering the sequence of surrounding words. To identify the optimal n-gram size, I systematically tested various sizes, ranging from 1 to 3, as well as combinations such as using both unigrams and bigrams. The tests were conducted with “random_state” set to 1, utilising the Scikit-learn Naïve Bayes and no text pre-processing. Consistently, the highest accuracy was achieved when incorporating all three of unigrams, bigrams and trigrams. The choice strikes a balance between providing context and the meaning of the individual words. A drawback, however, is that this approach also demands the most computational power, resulting in a longer training time for the models.

Feature selection

```python
# Preprocesses text according to parameters
def preprocess_text(doc, remove_stopwords=False, lemmatize=False, stem=False):
    words = doc.lower().split()
    if remove_stopwords:
        stop_words = set(stopwords.words('english'))
        words = [word for word in words if word not in stop_words]
    # (Can't have both lemmatization and stemming)
    if lemmatize:
        lemmatizer = WordNetLemmatizer()
        words = [lemmatizer.lemmatize(word) for word in words]
    elif stem:
        porter = PorterStemmer()
        words = [porter.stem(word) for word in words]

    # Joins the words again to create a preprocessed doc
    preprocessed_words = ' '.join(words)

    return preprocessed_words
```

Figure 1: Pre-process function code for feature generation

```python
# Uses tfidf for feature selection in the sparse matrix
def tfidf(feature_vec, N):

    # Compute Inverse Document Frequency (IDF)
    document_frequency = np.sum(feature_vec > 0, axis=0)
    idf = np.log((N) / (1 + document_frequency))

    # Calculate TF-IDF
    X_feature_vec_tfidf = feature_vec * idf

    return X_feature_vec_tfidf
```

Figure 2: TF-IDF function code

```python
# Feature set with stopwords removed, lemmatization and TFIDF (after vectorization)
feature_set_data = [preprocess_text(doc, remove_stopwords=True, lemmatize=True) for doc in combined_reviews]
```

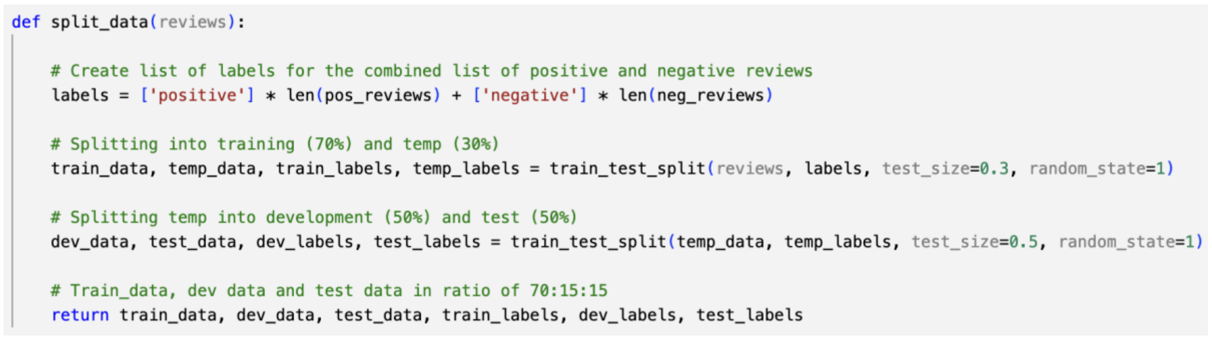
Figure 3: Example of code initialising a feature matrix

The "preprocess_text" function has been implemented with Boolean parameters for stopword removal, lemmatization and stemming, facilitating the generation of different feature sets. Additionally, a TF-IDF function has been implemented to apply feature weighting and normalisation to the pre-processed data.

For stopword removal, the nltk English stopwords package was used to eliminate common words with minimal semantic value, such as delimiters "a" and "the." To reduce the variations of words and reduce them to a common form, both WordNetLematizer and PoterterStemmer were utilised. Stemming involves using a set of language-specific morphological rules to truncate the prefixes or suffixes of a word whereas lemmatization analyses the context to identify the correct dictionary root of a word. The latter is more precise compared to stemming, however, a notable drawback is its slower computational speed. Finally, TF-IDF was employed for data normalisation, a process which essentially weights the significance of terms in a corpus.

Leveraging the implemented functions, five feature sets are generated for systematic testing. The initial three have either stopwords removed, stemming or lemmatization. The fourth feature set combines stopword removal with lemmatization, while the fifth feature set encompasses stopword removal, lemmatization and TF-IDF processing.

Data splits

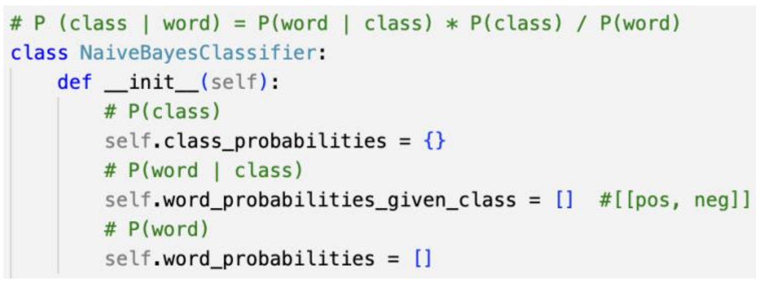


```
def split_data(reviews):

    # Create list of labels for the combined list of positive and negative reviews
    labels = ['positive'] * len(pos_reviews) + ['negative'] * len(neg_reviews)

    # Splitting into training (70%) and temp (30%)
    train_data, temp_data, train_labels, temp_labels = train_test_split(reviews, labels, test_size=0.3, random_state=1)

    # Splitting temp into development (50%) and test (50%)
    dev_data, test_data, dev_labels, test_labels = train_test_split(temp_data, temp_labels, test_size=0.5, random_state=1)

    # Train_data, dev data and test data in ratio of 70:15:15
    return train_data, dev_data, test_data, train_labels, dev_labels, test_labels
```

Figure 4: A function to split the data into training, development and testing sets

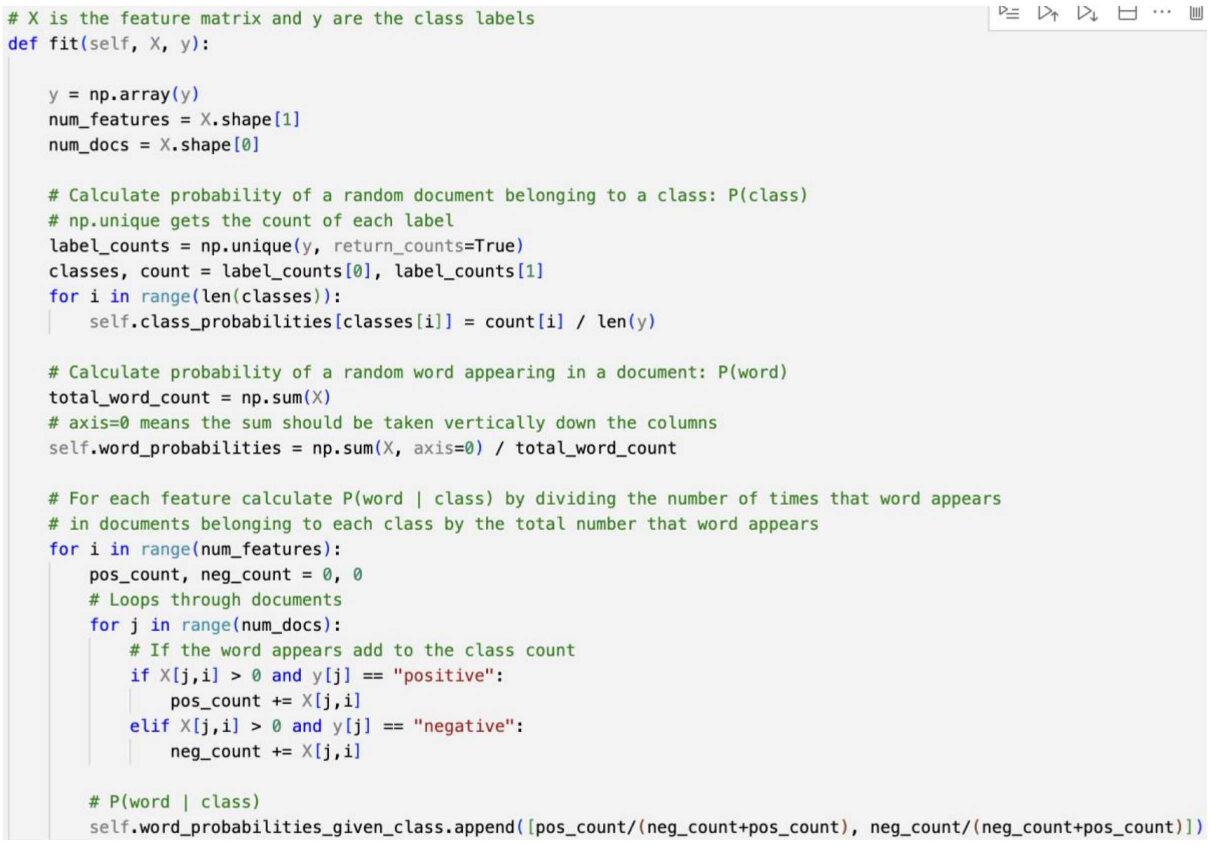
Using the Scikit-learn “train_test_split” function, the dataset has been partitioned into a 70:15:15 ratio for training, development, and testing, respectively. This data split aligns with the common practice for balanced datasets such as the one provided. The development set will serve as a validation set utilised for hyperparameter optimisation, while the test set, containing 600 previously unseen positive and negative reviews, is reserved for the ultimate evaluation of the model.

Naïve Bayes



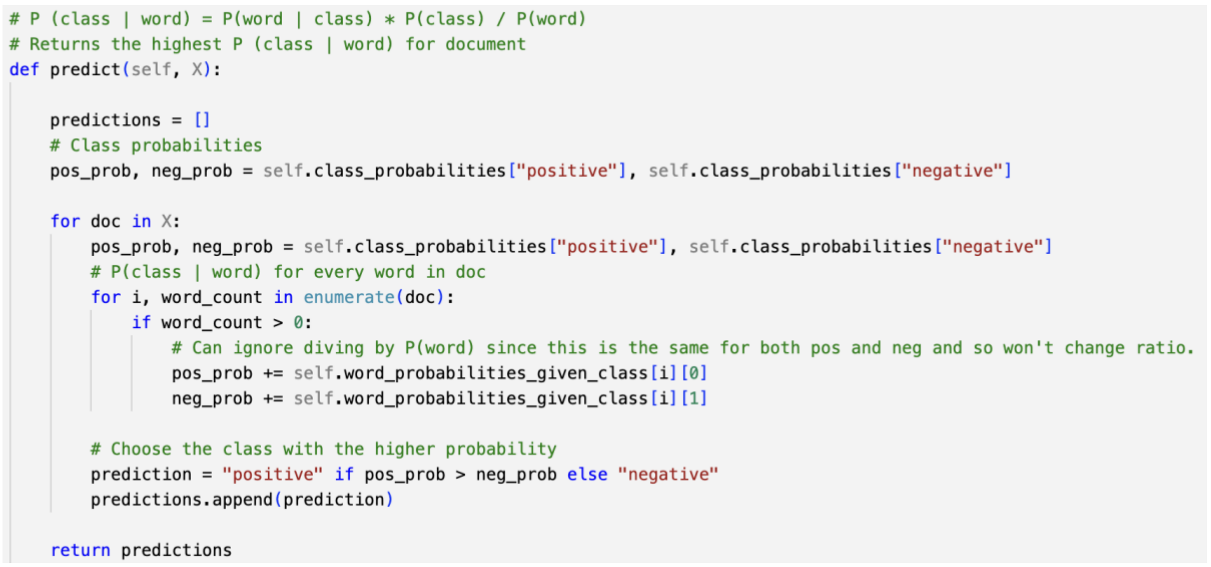
```
# P (class | word) = P(word | class) * P(class) / P(word)
class NaiveBayesClassifier:
    def __init__(self):
        # P(class)
        self.class_probabilities = {}
        # P(word | class)
        self.word_probabilities_given_class = []  #[[pos, neg]]
        # P(word)
        self.word_probabilities = []
```

Figure 5: Naive Bayes implementation class constructor



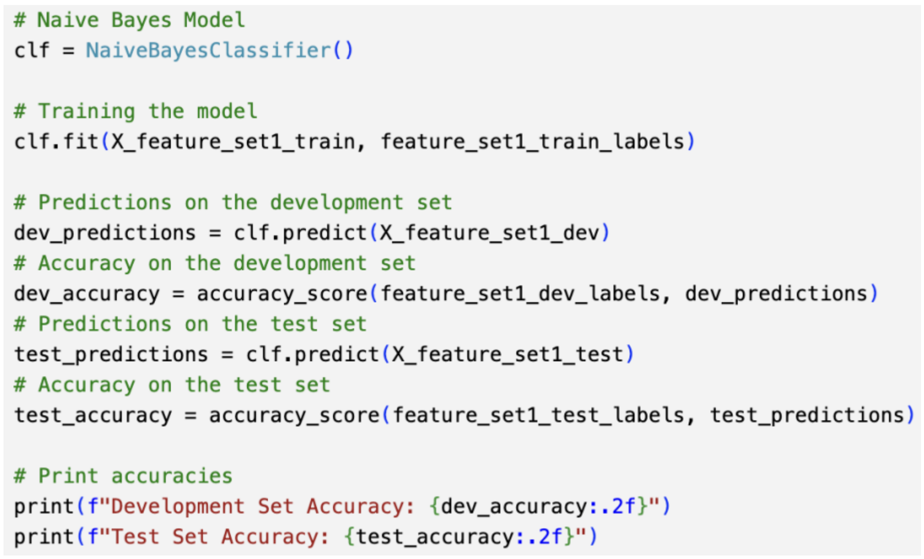
```python
# X is the feature matrix and y are the class labels
def fit(self, X, y):

    y = np.array(y)
    num_features = X.shape[1]
    num_docs = X.shape[0]

    # Calculate probability of a random document belonging to a class: P(class)
    # np.unique gets the count of each label
    label_counts = np.unique(y, return_counts=True)
    classes, count = label_counts[0], label_counts[1]
    for i in range(len(classes)):
        self.class_probabilities[classes[i]] = count[i] / len(y)

    # Calculate probability of a random word appearing in a document: P(word)
    total_word_count = np.sum(X)
    # axis=0 means the sum should be taken vertically down the columns
    self.word_probabilities = np.sum(X, axis=0) / total_word_count

    # For each feature calculate P(word | class) by dividing the number of times that word appears
    # in documents belonging to each class by the total number that word appears
    for i in range(num_features):
        pos_count, neg_count = 0, 0
        # Loops through documents
        for j in range(num_docs):
            # If the word appears add to the class count
            if X[j,i] > 0 and y[j] == "positive":
                pos_count += X[j,i]
            elif X[j,i] > 0 and y[j] == "negative":
                neg_count += X[j,i]

        # P(word | class)
        self.word_probabilities_given_class.append([pos_count/(neg_count+pos_count), neg_count/(neg_count+pos_count)])
```

Figure 6: Naive Bayes implementation fit method



```python
# P (class | word) = P(word | class) * P(class) / P(word)
# Returns the highest P (class | word) for document
def predict(self, X):

    predictions = []
    # Class probabilities
    pos_prob, neg_prob = self.class_probabilities["positive"], self.class_probabilities["negative"]

    for doc in X:
        pos_prob, neg_prob = self.class_probabilities["positive"], self.class_probabilities["negative"]
        # P(class | word) for every word in doc
        for i, word_count in enumerate(doc):
            if word_count > 0:
                # Can ignore diving by P(word) since this is the same for both pos and neg and so won't change ratio.
                pos_prob += self.word_probabilities_given_class[i][0]
                neg_prob += self.word_probabilities_given_class[i][1]

        # Choose the class with the higher probability
        prediction = "positive" if pos_prob > neg_prob else "negative"
        predictions.append(prediction)

    return predictions
```

Figure 7: Naive Bayes implementation predict method



```python
# Naive Bayes Model
clf = NaiveBayesClassifier()

# Training the model
clf.fit(X_feature_set1_train, feature_set1_train_labels)

# Predictions on the development set
dev_predictions = clf.predict(X_feature_set1_dev)
# Accuracy on the development set
dev_accuracy = accuracy_score(feature_set1_dev_labels, dev_predictions)
# Predictions on the test set
test_predictions = clf.predict(X_feature_set1_test)
# Accuracy on the test set
test_accuracy = accuracy_score(feature_set1_test_labels, test_predictions)

# Print accuracies
print(f"Development Set Accuracy: {dev_accuracy:.2f}")
print(f"Test Set Accuracy: {test_accuracy:.2f}")
```

Figure 8: Code to run the implemented Naive Bayes model. To run the Scikit-learn model, "NaiveBayesClassifier()" can be swapped with "MultinomialNB()"

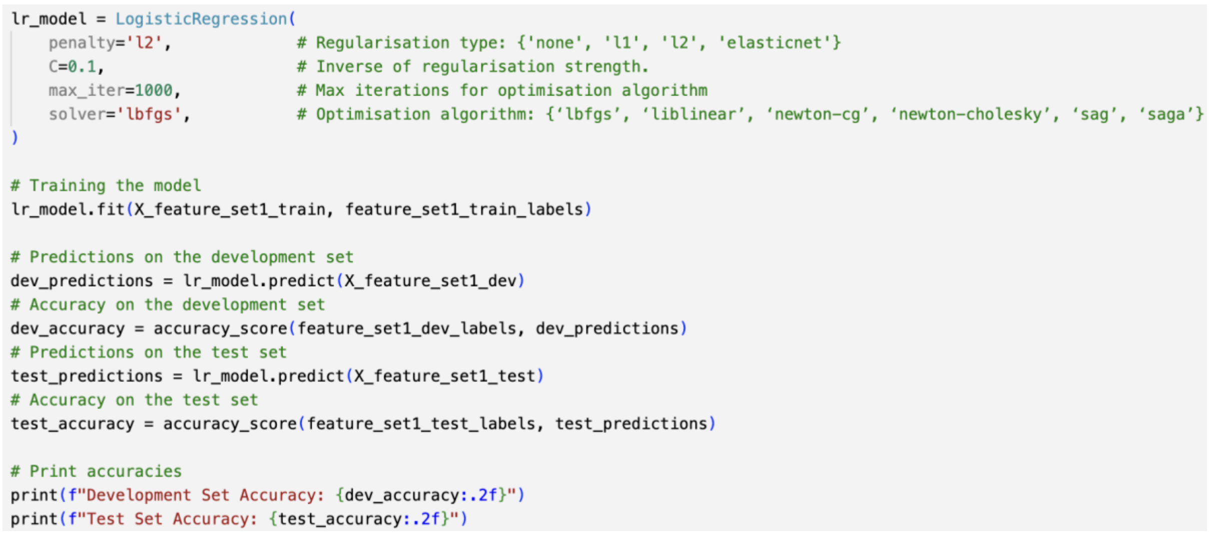
Feature set	My Naïve Bayes accuracy	Scikit-learn Naïve Bayes accuracy
1	0.85	0.83
2	0.84	0.83
3	0.84	0.83
4	0.84	0.83
5	0.84	0.84

Table 2: Average development set results taken from 3 different data split seeds

Following the assessment of the Naïve Bayes model on the development set, where the average score was computed across three distinct random seed data splits (rounded to two decimal places), the optimal feature set was identified as the first, which exclusively involved the removal of stopwords. The test results averaged to **0.83**. Subsequently, each feature set underwent evaluation using the Scikit-learn model where the most effective feature set was determined to be the one with stopwords removed, lemmatization and TF-IDF. This yielded an average test result of **0.82**.

Notably, the results differ slightly between implementations of the Naïve Bayes classifiers, with the Scikit-learn implementation comparatively not performing as well across all feature sets. These differences in results could be attributed to a distinction in the predict function. In the custom implementation, I chose to add the word probabilities given the class, contrary to using the Naïve Bayes independence assumption which entails multiplying them together. The reason for better results when doing this is due to numerical instability when multiplying extremely small numbers together.

SGD based classification and SVMs

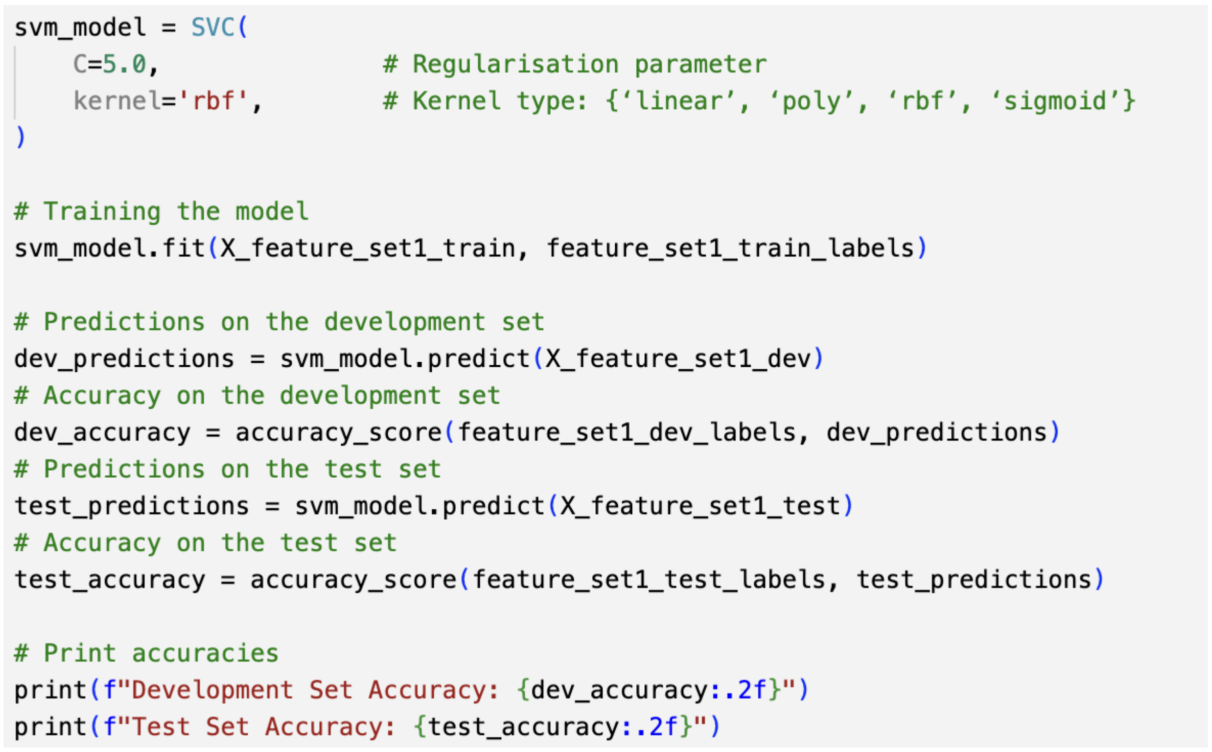


```
lr_model = LogisticRegression(
    penalty='l2',              # Regularisation type: {'none', 'l1', 'l2', 'elasticnet'}
    C=0.1,                     # Inverse of regularisation strength.
    max_iter=1000,             # Max iterations for optimisation algorithm
    solver='lbfgs',            # Optimisation algorithm: {‘lbfgs’, ‘liblinear’, ‘newton-cg’, ‘newton-cholesky’, ‘sag’, ‘saga’}
)

# Training the model
lr_model.fit(X_feature_set1_train, feature_set1_train_labels)

# Predictions on the development set
dev_predictions = lr_model.predict(X_feature_set1_dev)
# Accuracy on the development set
dev_accuracy = accuracy_score(feature_set1_dev_labels, dev_predictions)
# Predictions on the test set
test_predictions = lr_model.predict(X_feature_set1_test)
# Accuracy on the test set
test_accuracy = accuracy_score(feature_set1_test_labels, test_predictions)

# Print accuracies
print(f"Development Set Accuracy: {dev_accuracy:.2f}")
print(f"Test Set Accuracy: {test_accuracy:.2f}")
```

Figure 9: Logistic regression model code

Upon thorough evaluation of the feature sets on Scikit-learn’s logistic regression model, it was observed that feature set 5, configured with the parameters displayed in Figure 9, presented the best development set results. Assessed across the average of the same three random seeds, the outcome yielded a result of **0.85**. Similar to the results of the Naïve Bayes models, feature set 1, focussed simply on stopword removal, followed closely behind with a result of 0.84.

There were many different parameter options to configure the logistic regression model including changing the regularisation type, the inverse of the regularisation strength, the optimisation algorithm, and its maximum iterations. To experiment with the different options, systematic adjustments were made to the regularisation function, the inverse of its strength (C), and the optimisation algorithm, to evaluate 10 different combinations. The maximum iterations for the optimisation algorithm remained unaltered, as initial experimentation indicated it had negligible effect on results. The results indicate that the optimal hyperparameter configuration for the model entailed l2 regularisation, an inverse regularisation strength of 0.1, and the use of the lbfgs solver (equal to the performance of the liblinear function). Consequently, the evaluation of the feature sets, as previously outlined, was conducted using these hyperparameter values.



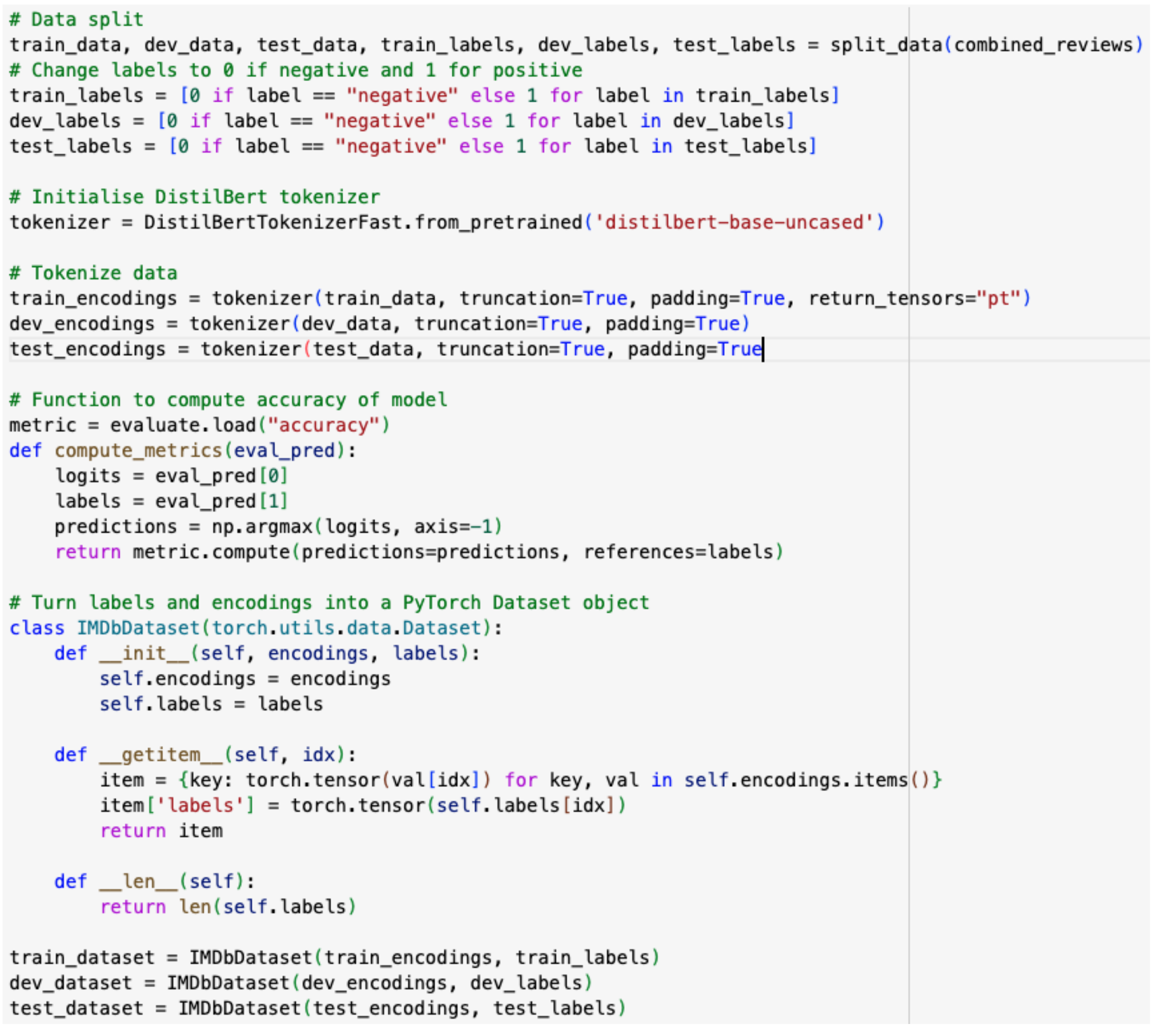
```
svm_model = SVC(
    C=5.0,                  # Regularisation parameter
    kernel='rbf',           # Kernel type: {‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’}
)

# Training the model
svm_model.fit(X_feature_set1_train, feature_set1_train_labels)

# Predictions on the development set
dev_predictions = svm_model.predict(X_feature_set1_dev)
# Accuracy on the development set
dev_accuracy = accuracy_score(feature_set1_dev_labels, dev_predictions)
# Predictions on the test set
test_predictions = svm_model.predict(X_feature_set1_test)
# Accuracy on the test set
test_accuracy = accuracy_score(feature_set1_test_labels, test_predictions)

# Print accuracies
print(f"Development Set Accuracy: {dev_accuracy:.2f}")
print(f"Test Set Accuracy: {test_accuracy:.2f}")
```

Figure 10: SVM model code

In the evaluation of the SVM model, it was determined that feature set with 5 (comprising of stopword removal, lemmatization and TF-IDF) performed best, with an average test set result of **0.84**. Again, the hyperparameter configuration process involved systematic adjustments to both the kernel and regularisation parameter.

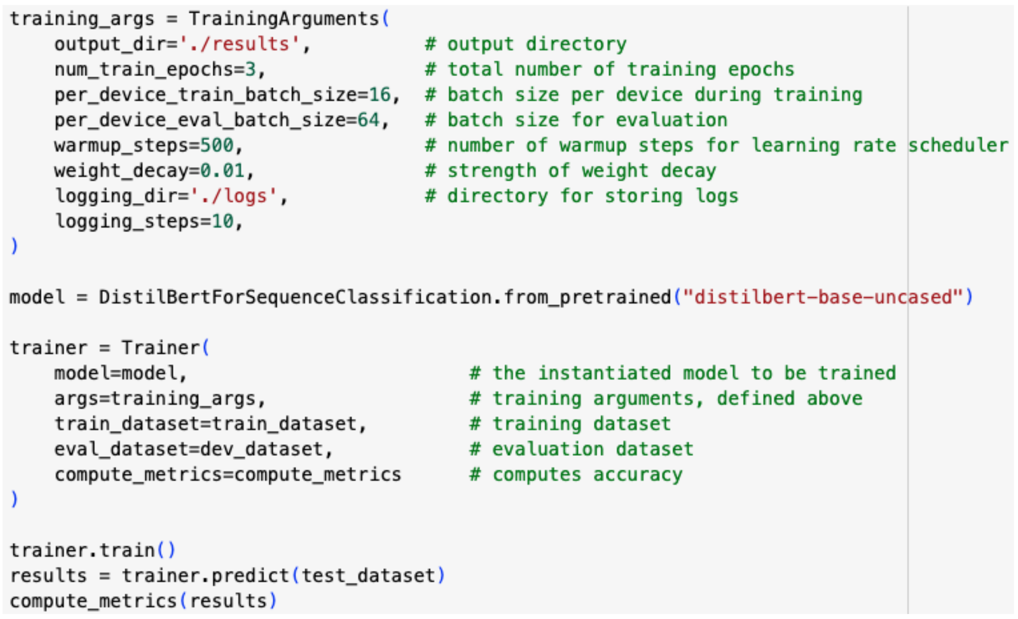
Incrementing the regularisation parameter while comparing kernel types revealed that the radial basis function (rbf) kernel with regularisation parameter of 5 performed optimally. Unlike the observations in the logistic regression model, some kernel types in the SVM model, such as the sigmoid kernel would completely “break” the model, resulting in a 0.5 accuracy, i.e., the equivalent of random guessing.

BERT



```
# Data split
train_data, dev_data, test_data, train_labels, dev_labels, test_labels = split_data(combined_reviews)
# Change labels to 0 if negative and 1 for positive
train_labels = [0 if label == "negative" else 1 for label in train_labels]
dev_labels = [0 if label == "negative" else 1 for label in dev_labels]
test_labels = [0 if label == "negative" else 1 for label in test_labels]

# Initialise DistilBert tokenizer
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')

# Tokenize data
train_encodings = tokenizer(train_data, truncation=True, padding=True, return_tensors="pt")
dev_encodings = tokenizer(dev_data, truncation=True, padding=True)
test_encodings = tokenizer(test_data, truncation=True, padding=True

# Function to compute accuracy of model
metric = evaluate.load("accuracy")
def compute_metrics(eval_pred):
    logits = eval_pred[0]
    labels = eval_pred[1]
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions, references=labels)

# Turn labels and encodings into a PyTorch Dataset object
class IMDbDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = IMDbDataset(train_encodings, train_labels)
dev_dataset = IMDbDataset(dev_encodings, dev_labels)
test_dataset = IMDbDataset(test_encodings, test_labels)
```

Figure 11: Bert data pre-processing and defining evaluation function



```
training_args = TrainingArguments(
    output_dir='./results',          # 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='./logs',            # directory for storing logs
    logging_steps=10,
)

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

trainer = Trainer(
    model=model,                         # the instantiated model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=dev_dataset,            # evaluation dataset
    compute_metrics=compute_metrics      # computes accuracy
)

trainer.train()
results = trainer.predict(test_dataset)
compute_metrics(results)
```

Figure 12: Training and testing the model

Following additional data pre-processing to ensure compatibility with BERT, training was conducted on both the cased and uncased versions of the model. Implementation guidance was provided by the tutorials available in the Hugging Face documentation (https://huggingface.co/). On seed 1 of the data split, the accuracy achieved using the cased version was **0.89**, while the uncased version yielded a slightly higher accuracy of **0.90**. The marginal difference suggests that in this task, retaining capital letters does not contribute significantly to the model's performance; in fact, it performed slightly worse. In tasks that require sensitivity to capital letters however, such as a Named Entity Recognition problem, the cased version would perform much better compared to the uncased counterpart.

As further elaborated in the discussion section, BERT significantly outperforms the previous models as it has already been pretrained on other data, allowing it to know which words are more closely associated with a positive and negative sentiment. Lastly, a brief exploration of hyperparameter optimisation was undertaken, as depicted in Figure 11 (which happen to coincide with the default settings). However, it is important to highlight that a comprehensive and systematic testing of hyperparameters unfortunately could not be carried out due to constraints of available time and the prolonged time required to train BERT, requiring use of the Google Colab GPU.

Discussion

Method	Test Split Result
Naïve Bayes (my implementation)	0.83
Naïve Bayes (Scikit-learn)	0.82
Logistic Regression	0.85
SVM	0.84
BERT	0.90

Table 3: Average results of the Test split using the different models

The overall results demonstrate accurate sentiment classification from all the models, ranging from 0.82 with Scikit-learn Naïve Bayes to 0.90 using the uncased version of BERT. Notably, among the models which were not pretrained, logistic regression exhibited the highest accuracy. While the specific choice of feature sets, elaborated upon later, played a significant part, one contributing factor to these differences could be due to the assumption of feature independence. Unlike with Naïve Bayes, logistic regression does not assume feature independence, potentially impacting its effectiveness.

The SVM model performed similarly to logistic regression, differing by only 0.01. This discrepancy might stem from random variations or due to differences in levels of hyperparameter optimisation, as an exhaustive exploration of all hyperparameters was not conducted.

Another intriguing observation lies in the variations in optimal feature sets across models. Notably, the performance disparities between models became evident only when using the optimal feature sets in conjunction with the optimal hyperparameters. For instance, certain configurations of the logistic regression model performed worse compared to Naïve Bayes, yet logistic regression outperformed both implementations of Naïve Bayes when using the optimal hyperparameters and features.

Specifically, the custom Naïve Bayes implementation and logistic regression model excelled when using the feature set with only stopwords removed. On the contrary, the Scikit-learn and SVM implementations performed best with stopword removal, lemmatization and TF-IDF. While it is not obvious what might cause these differences, the key discrepancy between the two feature sets is the presence of additional features in the set with only stopwords removed. Therefore, it is possible that different models excel in leveraging this supplementary information, albeit at the risk of overfitting. In contrast, the Scikit-learn and SVM models evidently benefit from carefully selected, normalised, and weighted features.

Lastly, it is important to mention training speed, where both Scikit-learn Naive Bayes and logistic regression could be trained relatively faster than the SVM and the custom Naïve Bayes implementation. BERT, on the other hand, took significantly longer to train compared to the other models. Finally, it is noteworthy that the training of the custom Naïve Bayes implementation could have certainly sped up through the implementation of vectorisation in all parts of the code.

Conclusions and Future Work

In conclusion, the choice of the best performing model depends on the specific priorities of the task. Logistic regression excelled in speed, making it the best option when efficiency is paramount. On the other hand, if accuracy is more important, BERT was the superior model. Moreover, further refinement through the creation and testing of more feature sets as well as further hyperparameter optimisation could have yielded even better results.

For future work, exploring embeddings to optimise feature extraction appears promising as the importance of data pre-processing was evident in this project. Additionally, combining high performing models such logistic regression and SVM into an ensemble method is an intriguing prospect. This could be done using techniques such as voting to combine the predictions of both models to make a final classification decision for example.

Overall, this project has not only deepened my understanding of the nuances involved in feature generation, selection, and extraction, but also provided valuable insights into the strengths and weaknesses of different models, resulting in a greater appreciation of the intricacies behind large language models.

Bibliography

Chiew, K.L.,, Tan, C.L.,, Wong, K.,, Yong, K.S.C., and Tiong, W.K., 2019. A New Hybrid Ensemble Feature Selection Framework for Machine Learning-based phishing detection system. *Information Sciences*, 484, pp.153–166.

Gonçalves, P.,, Araújo, M.,, Benevenuto, F., and Cha, M., 2013. Comparing and combining sentiment analysis methods. *Proceedings of the first ACM conference on Online social networks*.

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

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

Yilmaz, B., 2023. *Sentiment Analysis Methods in 2023: Overview, pros & cons*. [online] AIMultiple. Available from: https://research.aimultiple.com/sentiment-analysis-methods/ [Accessed 11 Dec. 2023].