⟨Natural Language Processing Assignment 1⟩

⟨Jamal Akhras⟩

⟨Computer Science and AI⟩
The University of Bath
⟨2023/2024⟩

⟨Semantic Analysis and Classification⟩

Submitted by: ⟨Jamal Akhras⟩

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

⟨This project focuses on sentiment analysis using IMDb reviews, aiming to create a robust application for polarity detection in textual content. The process involves data preprocessing, data analysis, feature generation, and feature selection. Model selection includes Naive Bayes, SGD-based classification, SVMs, and BERT. The study evaluates the impact of hyper-parameter optimization on model performance and compares custom implementations with scikit-learn's implementations. The assignment requires systematic feature generation using n-grams, feature selection employing stopwords, lemmatization, and TFIDF, as well as data splits into training, development, and test sets.

The experiments include implementing Naive Bayes from scratch, evaluating scikit-learn's implementation, and assessing SGD-based classification and SVMs with hyper-parameter optimization. Additionally, the study explores BERT-based classification, considering both cased and uncased models. The discussion analyzes and compares results from each method on the test split, emphasizing the performance factors influencing each model. Concluding insights and future work considerations are provided to enhance the understanding and potential improvements for sentiment analysis models..⟩

Contents

Acknowledgements

This research made use of Hex, the GPU Cloud in the Department of Computer Science at the University of Bath.

Chapter 1

Introduction

Sentiment analysis is a growing field at the intersection of linguistics and computer science that is based around automatically determining the sentiment/meaning behind a piece of text. Seeing as the evolution of the internet has given people the unprecedented ability to portray and express their views via online mediums, including social media, blogs, and as going to be focused on in this project, movie reviews. Therefore, it plays an important part of Natural Language Processing (NLP), a field of computer science and artificial intelligence that mainly deals with the interaction between humans and computers.Medhat, Hassan and Korashy (2014) The importance of semantic analysis applies to every sector that has an online presence, as it provides an efficient and practical way to read through the massive amounts of reviews online and develop an understanding of what needs to be done to improve the service provided. However, the issue with reviewing text to identify sentiment is the inability to portray tone. A common issue faced in semantic analysis is that not all the reviews are legitimate, this can manipulate the application into misclassifying the sentiment of a piece of text and providing inaccurate results.Medhat, Hassan and Korashy (2014) Some examples of the difficulties faced in sentiment analysis include:

- Ambiguity and Context Dependency
  Ambiguous language is often used while writing reviews and rely heavily on the context of the sentence, however, words and phrases can have different meanings based on its context and this makes it challenging to accurately determine sentiment.
- Negation and Irony
  Negation and irony reverses the polarity of a statement, accurately identifying and interpreting these expressions requires an immense amount of training data to achieve a deep understanding of context and the nuances of language.
- Subjectivity and Diversity
  Sentiment, although can be arguably considered objective, are highly subjective and are different based on demographic, cultures, and groups of individuals. What one might find positive can potentially be interpreted as negative by another.
- Sarcasm
  Another difficulty faced in semantic analysis is sarcasm, since the expressions conveys sentiments opposite to what is intended.
- Data Imbalance

A common issue in machine learning is when the data set does not have an even proportion of examples among the classes. Indirectly biasing the model towards the majority class.

- Temporal Dynamics
  As the world changes so do sentiments, so a phrase that was once positive could become negative and as such constant retraining of the model would prove to be a difficulty.

Addressing these difficulties often involve using a combination of multiple advanced NLP techniques and machine learning models to adapt to the evolving trends.
There are many methods used to approach the issue of sentiment analysis, these include:

- Rule-Based methods **Lexicon-Based Approaches** these methods rely on sentiment lexicons or dictionaries that contain words paired with their respective sentiments. This is a very primitive approach and due to the large nature of languages and that words can change meaning depending on context make this method extremely inefficient
- Machine Learning Approaches
  - Supervised Learning
    This is done by training a model based on a labeled data set (in this case reviews paired with sentiment). Algorithms that fall under this category include: Support Vector Machines(SVM), Naive Bayes, and Decision treesCunningham, Cord and Delany (2008)
  - Unsupervised Learning
    In unsupervised training the models do not need labeled training data, instead the data is analyzed and patterns/correlations are found. Common techniques are clustering(K-means) and topic modelling can be used to identify patterns and meanings of text. Barlow (1989)
- Deep Learning
  - Recurrent Neural Networks (RNN)
    RNNs work by using the output of an older layer in a newer layer to incorporate a sense of memory into the algorithm, a popular example of this would be long-short term memory (LSTM) networks. This is useful in sentiment analysis because they can capture sequential dependencies in text. Salehinejad et al. (2018)
  - Convolutional Neural Networks(CNN)
    CNNS are used to extract local patterns from input data using kernels that scan through the data set and only take into account neighboring items of data. They can be used for sentiment analysis to capture relevant features from text.Ouyang et al. (2015)
- Hybrid Approaches
  Using machine or deep learning methods alongside rule based methods can improve the accuracy of sentiment analysis systems. An example of this can be using a rule based system for certain cases and machine learning models can provide the overall sentiment classification. Appel et al. (2016)
- Sentiment Analysis using pre-trained models
  There are language models available online that have already been trained which allow the

users to just implement. Popular examples include Bidirectional Encoder Representations from Transformers (BERT), or Generative Pre-trained Transformer(GPT). These pre-trained models already have contextual understanding and can be altered to be used for specific sentiment analysis tasks Devlin et al. (2019)

- Natural Language Processing Techniques
  - Text preprocessing
    Tokenization involves breaking down all the text into individual words or "tokens". Lower casing is useful to provide uniformity among the text, and removing stopwords helps reduce the amount of text to process by eliminating words like ("the", "and", "or") and others that do not relate to sentiment
  - Bag of Words
    Bag of words entails representing the text as an unordered set of words, where it is then represented as a vector, where each element corresponds to the frequency of each word.
  - Term Frequency Inverse Document Frequency (TF-IDF)
    This method assigns weights to the words in a document based of their frequency relative to their frequency across all documents, where documents is all the pieces of text to analyze. This ensures that the focus is on words that are more unique to a document and are more likely to carry the sentiment with them.
  - N-grams
    This technique involves grouping words that are adjacent to capture context. Uni-grams would take each word, bi-grams would take every 2 adjacent words, tri-grams take 3 adjacent words and so on.Yi et al. (2003)

None of the lists are exhaustive, as they prioritize the more popular and relevant methods to this project.

This assignment use some of the methods mentioned above, which will be explained more in depth later in the assignment, to generate example responses using the provided data and compare the results, namely: Naive Bayes, SGD, SVM, BERT, and a combination of the different NLP techniques mentioned to improve performance. Additional methods were used within the custom implementation of Naive Bayes, including using Logarithms and La-Place smoothing.

- Logarithms
  Naive Bayes works by using Bayes theorem, the theorem is computed as the product of the probabilities of individual features(words/tokens) given the class. For a given class 'C' and a document 'D', the probability P(C|D) is proportional to the product of P(D|C) also referred to as the "likelihood" and P(C) also referred to as the prior. This can be expressed as

  $$P(CD)P(DC)P(C)$$

  Because of the nature of the algorithm and multiplying many small probabilities, this often leads to issues of numerical instability. Taking the logarithm helps mitigate this because of the logarithm property that the logarithm of a product is equal to the sum of logarithms. The formula becomes

  $$log(P(CD))log(P(DC)) + log(P(C))$$

Doing so allows the multiplication to be converted to addition, which is computationally more stable. The main use of this is when the data set is large or when there are a large number of features.

- Laplace Smoothing
  Another common issue with Naive Bayes is when the calculated estimated likelihood of a feature. Recalling that the probabilities are calculated based on the frequency of the features, however, if a feature is not in the training data, the probability is zero. Laplace smoothing brings in a small constant to all the feature counts, ensuring that there is no probability left as exactly zero. The formula for Laplace Smoothing probability estimation is:

$$P(wordclass) = (\frac{count(word, class) + k}{count(class) + k * |V|}$$

  Where count(word,class) is the number of occurrences of the word in the training data for the given class, count(class) is the total number of words in the training data, |V| is the number of unique words across all classes, and k is the Laplace smoothing parameter (usually 1). By adding "k" to the numerator and denominator , this ensures no probability becomes zero, and the probabilities are then used in the Naive Bayes Classifier allowing for more robust estimates.Ramadhani, Indriani and Nugrahadi (2016)

Chapter 2

Related Work

A more detailed description for each of the methods/algorithms used are provided alongside their code implementation and the results achieved from them.

Chapter 3

Experiments and Results

This section explores the different methods mentioned above and the collection of techniques used to achieve the results

3.1 Feature Generation using n-grams

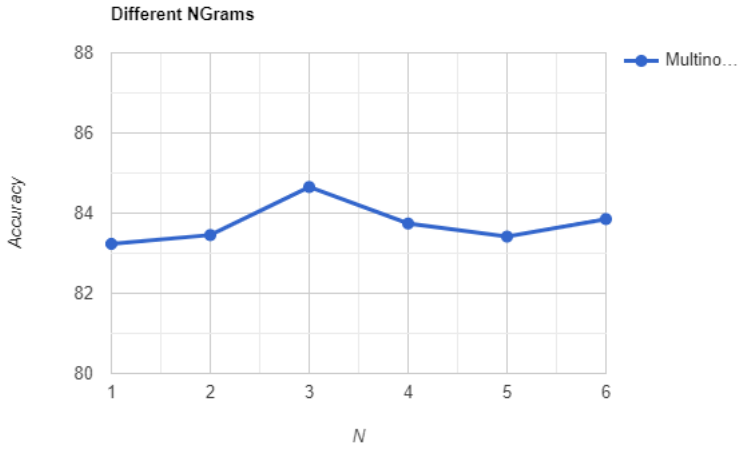
```
def extractNgrams(self,tokens):
ngrams = []
for i in range(self.ngramRange[0], self.ngramRange[1]):
    ngrams.extend(["␣".join(tokens[x: x + i ]) for x in
       range(len(tokens) - i + 1)])
return ngrams
```

The code above is an implementation of ngrams where the number of adjacent words to be considered, or 'n', is set by the user when instantiating the object to said class.
Selecting the right value of n in ngrams means finding the right balance between capturing enough context and avoiding too much computational complexity. The ideal choice on n depends on the task and the amount of context required. The two main methods for tuning n are empirical testing and grid search. Empirical testing is when the user manually tries different values of n and compares results at the end, a grid search on the other hand, does so automatically by testing all the different values of n in a range and identifying the most successful, below is the code implementation.

```
def selectNgramSize(data, labels, ngram_sizes):
    bestngramSize = None
    bestAccuracy = 0.0

    for n in ngram_sizes:
        # Create CountVectorizer with the current n-gram size
        vectorizer = CountVectorizer(ngram_range=(1, n))
        X = vectorizer.fit_transform(data)

        # Use Naive Bayes classifier for demonstration purposes
        classifier = MultinomialNB()

        # Perform cross-validation
```

```
            scores = cross_val_score(classifier, X, labels, cv=5,
                scoring='accuracy')
            meanAccuracy = np.mean(scores)

            # Update best parameters if the current n-gram size has
                higher accuracy
            if meanAccuracy > bestAccuracy:
                bestAccuracy = meanAccuracy
                bestngramSize = n

        return bestngramSize, bestAccuracy
```

The code only changes the optimal value of n if it outperforms the mean of all the other attempts. The tests were done using sci-kit learns built in version of Multinomial Naive-Bayes to ensure optimal and consistent performance. On the given data, after splitting, below is the graph of the results achieved.



Based on the results it can be said that for the given data, a value of n equal to three is the most optimal.

3.2 Feature Selection

All the separate techniques are applied in the preprocessing class in the accompanying code folder. It is also worth noting that all the methods are applied after tokenization, which is the separation of the sentence into individual words/tokens.

- Stopwords
  Stopwords are words that do not carry any meaning or do not bear much in relation to the sentiment of the text, this refers to words such as ("and", "or", "the"). Additionally, they reduce the complexity of the task by reducing the number of features. Models trained without stopwords are also more generalizable since they are common across most documents. This projects implementation of removing stopwords is as follows:

```
        def removeStopwords(self, review):
            filtered_words = [word for word in review if word
                not in stopWords]
```

```
        return filtered_words
```

Where "stopWords" is a reference to the nltk stopwords corpus.

- Lemmatization
  Lemmatization is the process of reducing words back to their base/root form, known as lemmas. Lemmatization is done to standardize words, ensuring that different forms of a word are treated as a single base form, this helps reduce the dimensionality and improve the accuracy of text analysis tasks. Rather than create a lemmatization algorithm, this project utilizes the "nltk.stem WordLemmatizer" to ensure more consistent and efficient results.

```
def doLemmatization(self, review):
    out = review
    for word in out:
        each = lemmatizer.lemmatize(word)
    return out
```

  Where "lemmatizer" is instantiated as:

```
lemmatizer = WordNetLemmatizer()
```

- Stemming
  Stemming is very similar to lemmatization in the sense that it is a method of reverting a word to a base/root form, known as a stem. The difference is that stemming removes prefixes and suffixes form the word, allowing words with similar meanings to be treated as the same word. Stems are not always valid words, but they capture the core meaning and help with semantic analysis. The stemming method used in the project is the nltk adaptation of the porter stemmer, one of the most widely used stemming algorithms, as shown in the code snippet.

```
def doStemming(self, review):
    out = review
    for word in out:
        word = ps.stem(word)
    return out
```

  Where ps is the nltk porter stemmer, as shown below

```
ps = PorterStemmer()
```

  The drawback of stemming is that it may produce stems that are not valid words, and as a result the loss of linguistic precision may lead to worse accuracy during classification

- Term Frequency Inverse Document Frequency (TFIDF)
  TFIDF is the ratio of the number of times a term appears in a document relative to a collection of documents or corpus. It is used to evaluate the significance of words or terms in a document.

  Term Frequency(TF) measures the frequency of a word within a document, it is a ratio of the number of times that word appears to the total number of terms, this emphasizes the words that appear frequently in a document.

$$TF(t,d) = (\frac{numberofoccurencesoftermtindocumentd}{totalnumberoftermsindocumentd})$$

Inverse Document Frequency(IDF) assesses the importance of a term across all the documents(corpus). It is calculated as the logarithm of the ratio of the total number of documents to the number of documents containing the term. The purpose is to decrease the significance of words that are common.

$$IDF(t, D) = log(\frac{totalnumberofdocumentsinthecorpusN}{Numberofdocumentscontainingtermt + 1}$$

similar to Laplace smoothing, 1 is added to the denominator to avoid division by 0 in the case where the term does not appear in any other documents.

Calculating TF-IDF is done by multiplying TF and IDF together for each term in a document, the result is how important each term is in each document.

$$TF - IDF(t, d, D) = TF(t, d) * IDF(t, D)$$

The TF-IDF score is higher when a term is common in a document and is relatively uncommon across the entire corpus.

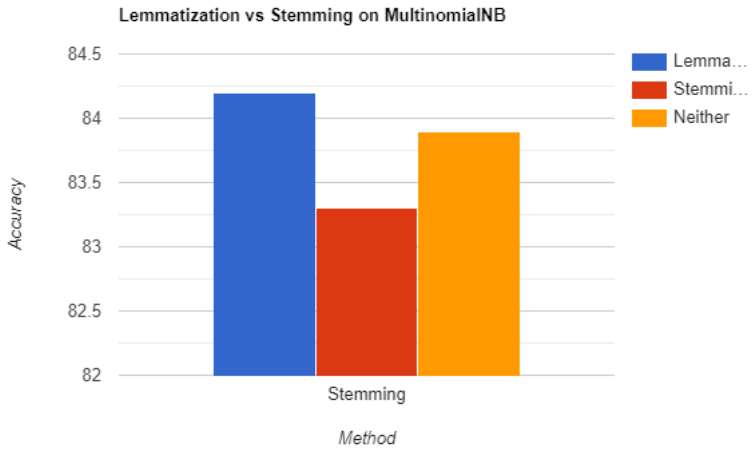
Rather than using an inbuilt version of TFIDF below is the code snippet of the projects implementation.

```
class tfidfVectorizer:
def __init__(self):
    self.vocab = set()
    self.wordCount = defaultdict(int)
    self.docCount = defaultdict(int)

def fitTransform(self, documents):
    for document in documents:
        tokens = self.preprocess(document)
        ngrams = self.extractNgrams(tokens)
        self.vocab.update(ngrams)
        for ngram in set(ngrams):
            self.wordCount[ngram] += ngrams.count(ngram)
        for ngram in set(ngrams):
            self.doc_count[ngram] += 1

    totalDocs = len(documents)
    featureVectors = lil_matrix((totalDocs,
       len(self.vocab)))

    for i, document in enumerate(documents):
        tokens = self.preprocess(document)
        ngrams = self.extractNgrams(tokens)
        for ngram in set(ngrams):
            if ngram in self.vocab:
                idx = list(self.vocab).index(ngram)
                tf = ngrams.count(ngram) / len(tokens)
                idf = np.log(totalDocs /
                   self.docCount[ngram])
```

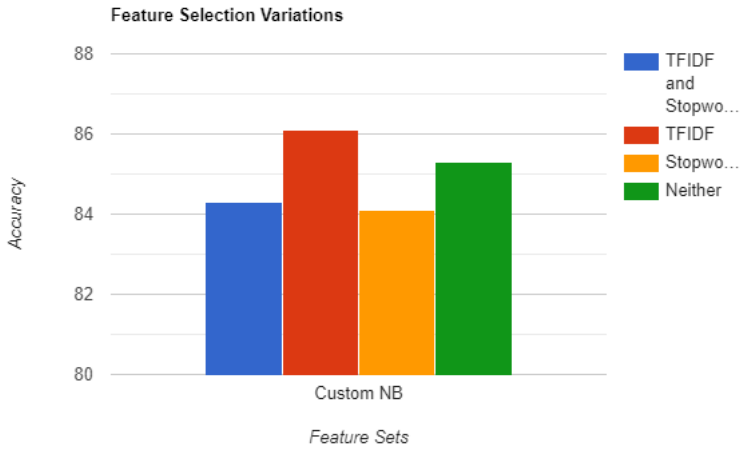
```
                tfidf = tf * idf
                featureVectors[i, idx] = tfidf

        return featureVectors
```

Note that scipy sparse matrices are used for their efficiency of representing text data. When converted to numbers text data often involved many dimensions, and sparse feature spaces where the majority of entries are zero. The benefits of using them are their memory efficiency, computational efficiency, reduction of dimension, and scalability.

Initially a test was done to determine whether lemmatization or stemming was to be used, and it was found that lemmatization outperforms stemming.



Once it was decided that 3-grams and lemmatization were to be used, a comparison was made between TFIDF and removing stopwords, testing whether it was the best to have both, either, or none.



Based on the results it can be concluded that tokenizing the text, having 3-grams, lemmatizing, **not** removing the stopwords, and using TFIDF would provide the best results.

3.3 Data Splits

It is imperative that the data is split before running any sort of machine learning algorithm, this is done to evaluate the performance of the model on unseen data, avoid over fitting, and tuning the hyper parameters.

In this project the data is split into 3 sets, a training, development, and test set. The training set is used to train the model. the development set is used during the training process to tune hyperparemeters, this is done to improve generalization. Finally, the test set is completely unseen by the model before and is used a the very end of development to evaluate the performance of the model on unseen data.

The project uses the sk-learn train_test_split function to achieve this. It was used twice, once to generate the train and test split, and the second to split the train into train and development/validation as shown in the code snippet below.

```
(x_train, x_test, y_train, y_test) =
    train_test_split(reviews, labels, test_size = 0.2)
(x_train, x_val, y_train, y_val) =
    train_test_split(x_train, y_train, test_size = 0.2)
```

this leaves the model with 800 reviews to test against, 620 reviews for development and 2560 reviews to train with. No specific testing was done to decide the sizes of each.

3.4 Naive Bayes

Naive Bayes is a group of probability based classification algorithms that are based on Bayes' theorem, with the "naive" assumption that the features are independent of each other. Although simple Naive Bayes classifiers often perform well and are commonly used for various applications, some of which are, natural language processing, spam filtering, document classification, and sentiment analysis.

- Bayes Theorem
  As mentioned earlier in the report, Bayes theorem describes the probability of an event based on the past knowledge of conditions related to the event. It is shown by:

  $$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

  In classification, the theorem is used to calculate the probability it belongs to a class given the observed features.

- Naive Assumption
  The assumption is that the features used to describe an instance are conditionally independent, given the class label. This assumption helps simplify and make the computation more manageable, particularly with high dimensional data. This assumption is often unrealistic in practice, however, even when the assumption is broken the classifiers still perform well.

- The different types of Naive Bayes Classifiers
  Depending on the type of data and distribution, different variants of the Naive Bayes classifiers would have to be used, common types are:

  - Gaussian Naive Bayes
    Makes the assumption that features are normally distributed
  - Bernoulli Naive Bayes
    Best suited for binary feature data, and often used in text classification with binary bag-of-words features.
  - Multinomial Naive Bayes
    Used for discrete data, most commonly in text classification with term frequencies.

  Given the data features and requirements of this assignment Multinomial Naive Bayes (Multinomial NB) fits best.

- Classification Process
  Given a set of features $X = x1, x2, ..., xn$ and a set of class labels $C = c1, c2, ..., cn$, the process finds the class ci that maximizes the posterior probability($P(Ci|X)$) using Bayes' theorem:

$$P(CiX)P(XCi) * P(Ci)$$

  The class with the highest posterior probability is the predicted class

  Even though Naive Bayes classifiers do not capture complex relations in the data, this is made up for by their simplicity, efficiency, and effectiveness.

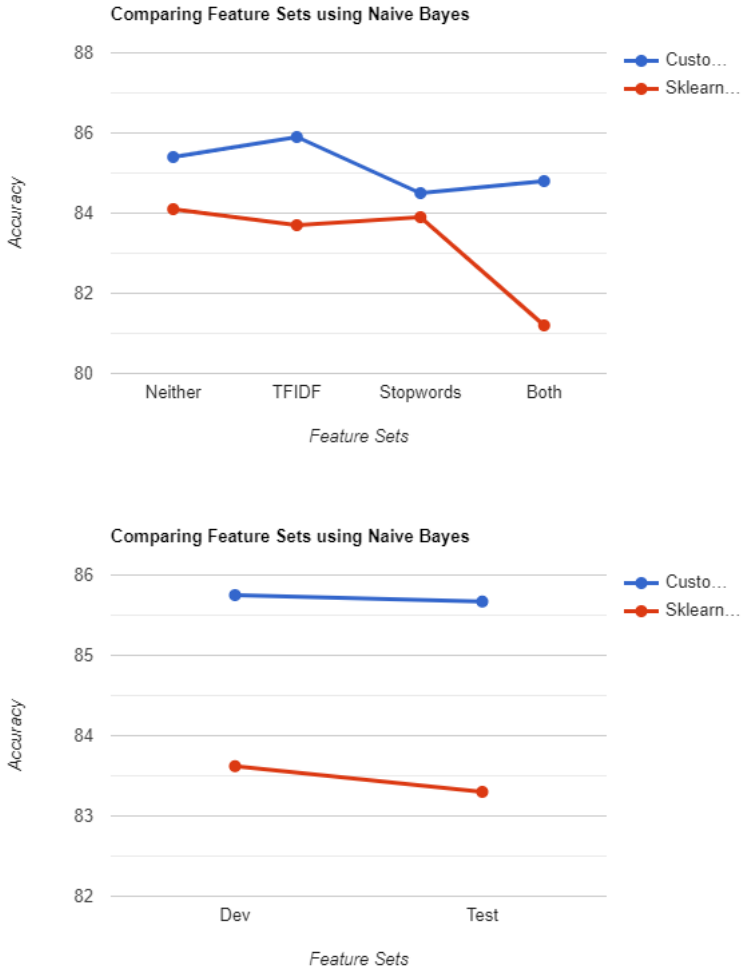
Below is the custom implementation of Multinomial Naive Bayes:

```
class NaiveBayesClassifier:
    def __init__(self, alpha=1.0):
        self.class_probs = None
        self.word_probs = None
        self.vocab = None
        self.alpha = alpha
        self.lemmatizer = WordNetLemmatizer()

    def preprocess(self, doc):
        doc = re.sub("[^A-Za-z0-9 ]", "", doc)
        tokens = word_tokenize(doc.lower())
        tokens = [self.lemmatizer.lemmatize(token) for token in
            tokens if token not in ENGLISH_STOP_WORDS]
        return tokens

    def train(self, X, y):
        self.class_probs = {label: np.sum(y == label) / len(y) for
            label in set(y)}
        self.word_probs = {}
        for label in set(y):
            label_docs = X[y == label]
            total_words = np.sum(label_docs)
            self.word_probs[label] = (np.sum(label_docs, axis=0) +
                self.alpha) / (total_words + self.alpha *
                len(self.vocab))

    def predict(self, X):
```

```
        # Calculate log probabilities for each class
        log_probs = {label: np.log(self.class_probs[label]) +
            X.dot(np.log(self.word_probs[label]).T) for label in
            self.class_probs}

        # Predict the class with the highest log probability
        predictions = np.argmax(np.vstack(log_probs.values()),
            axis=0)
        return predictions

    def evaluate(self, X, y):
        # Make predictions on the entire dataset
        predictions = self.predict(X)

        # Evaluate accuracy
        accuracy = accuracy_score(y, predictions)
        print(f"Accuracy: {accuracy:.2f}")
```

This assignment also leverages sk learns in-built Multinomial NB classifier, which was implemented as follows:

```
class MultinomialNaiveBayes:
    def __init__ (self):
        self.vectorizer = CountVectorizer()
        self.model = MultinomialNB()

    def train(self, xTrain, yTrain, xTest, yTest):

        self.model.fit(xTrain, yTrain)
        yPred = self.model.predict(xTest)

        accuracy = accuracy_score(yTest, yPred)
        print(f"Validation Accuracy: {accuracy}")

    def predict(self, xTest):
        predicitons = self.model.predict(xTest)
        return predicitons
```



After testing it was found that the custom implementation of Naive Bayes outperformed the inbuilt version consistently and across different data splits. It can be seen that TFIDF and stopwords do not work well together in this case, this is due to how TFIDF works and the removal of less significant words alters the algorithms view of more important words since it works based on the relation between term and document frequencies. The results of the different methods will be elaborated later in the report.

3.5 Stochastic Gradient Descent and Support Vector Machine based Classification

Stochastic Gradient Descent(SGD) is a variant of regular gradient descent algorithm, it is mainly used to train machine learning models. For classification SGD is often applied to logistic regression and linear support vector machines (SVM), and others. A brief overview of SGD is as follows:

- Objective/Cost function
  SGD is an algorithm that works by minimizing a cost function, in classification the cost function measures the difference between the predicted output and the true outputs
- Stochastic Gradient Descent

Regular gradient descent processes the entire training dataset to compute the gradient of the cost function with respect to the model parameters, while SGD only processed one randomly chosen training instance, or batch of instances at a time, to estimate the gradient

- Updating Parameters
  The models parameters(weights and biases) are updated based on the negative gradient of the cost function with respect to the parameters, the rule can be show by:

$$\theta = \theta - J(; xi, yi)$$

  where is the learning rate, $J(; xi, yi)$ is the cost associated with the i-th training instance and $J(; xi, yi)$ is the gradient of the cost with respect to θ

- Learning Rate
  The learning rate is a hyperparameter that controls the size of the steps taken during each update. It influences the convergence and stability of the optimization process, a small learning rate will take longer to find convergence while a large learning rate may overshoot the optimal parameters and lead to instability.

- Batch Size
  SGD can also be split based on the size of the batches used in each iteration. When the batch size is 1, it is referred to as true SGD. Batch sizes larger than 1 give a balance between the efficiency of batch gradient descent and the faster convergence of true SGDs.

- Epoch
  An epoch is one complete pass through the entire training dataset.

- Convergence
  Convergence is SGD is assessed based on the reduction of the value of the cost function or on the improvement of the classification accuracy. Training usually stops when convergence is reached, commonly a set number of epochs or satisfactory accuracy.

- Regularization
  Regularization terms, such as L1(lasso) or L2(Ridge) can be added to the cost function to prevent overfitting during optimization.

SGD is useful when dealing with large datasets or when memory constraints limit the use of the entire dataset in each iteration. It also has an element of randomness in the parameter updates, which help escape local minima and more efficiently reach convergence. Prasetijo et al. (2017)

3.5.1 Logistic Regression

Logistic Regression is a statistical model used for binary classification that calculates the probability of an instance belonging to a certain class.Nick and Campbell (2007) Some key concepts with logistic regression are:

- Logistic Function (Sigmoid)
  Logistic regression applies the logistic function, known as the sigmoid function, to map

a linear set of input features to a value between 0 and 1. The sigmoid function is:

$$(z) = \frac{1}{1 + e^{-}z}$$

where z is a linear combination of the input features and model parameters

- Model Hypothesis
  the logistic regression hypothesis function is expressed as

$$h(x) = ({}^{T}x)$$

  where $h(x)$ is the predicted probability that y = 1, given input features x. θ represents model parameters(coefficients) and x is the input feature vector

- Decision Boundary:
  The decision boundary is a hyperplane of dimensions - 1 that separates the instances of the 2 classes, all instances to one side are classed as one thing and the other side as the other. The hyperplane is drawn using the coefficients θ

- Training
  The training process is centered around parameter estimation. This is done using a process called maximum likelihood estimation. The goal is to find parameter values that maximize the likelihood of the observed data given the model.

- Cost Function
  The cost function is derived from the likelihood function and its designed to penalize variation between predicted probabilities and actual class labels. A common choice for the loss function is binary cross-entropy loss.

The following code snippet is an implementation of sklearn logistic regression:

```
    class LogisticRegressionClassifier:
def __init__(self, penalty='l2', dual=False, tol=1e-4, C=1.0,
   fit_intercept=True,
            intercept_scaling=1, class_weight=None,
               random_state=None, solver='lbfgs',
            max_iter=100, multi_class='auto', verbose=0,
               warm_start=False, n_jobs=None,
            l1_ratio=None):

    self.model = LogisticRegression(
        penalty=penalty, dual=dual, tol=tol, C=C,
           fit_intercept=fit_intercept,
        intercept_scaling=intercept_scaling,
           class_weight=class_weight,
        random_state=random_state, solver=solver,
           max_iter=max_iter,
        multi_class=multi_class, verbose=verbose,
           warm_start=warm_start,
        n_jobs=n_jobs, l1_ratio=l1_ratio
    )
    self.trained = False
```

```
    def train(self, X_train, y_train):
        self.model.fit(X_train, y_train)
        self.trained = True

    def predict(self, X_test):
        if not self.trained:
            raise ValueError("Classifier␣has␣not␣been␣trained.␣Train␣
                the␣model␣before␣making␣predictions.")
        return self.model.predict(X_test)

    def evaluate(self, X_test, y_test):
        if not self.trained:
            raise ValueError("Classifier␣has␣not␣been␣trained.␣Train␣
                the␣model␣before␣evaluating.")

        predictions = self.predict(X_test)
        accuracy = accuracy_score(y_test, predictions)
        report = classification_report(y_test, predictions)

        return accuracy, report
```

Note all the hyper parameters are made visible for tuning later on.

3.5.2 Support Vector Machines(SVM

SVMs are popular supervised machine learning tools that find and manipulate a hyperplane in feature space that separates data into different classes. The main ideas of SVM classification are:

- Objective Function
  The objective function aims to maximize the margin between the decision boundary (hyperplane) and the support vectors while minimizing the classification error, essentially placing the boundary as far from either nearest member of each class as possible. For soft margin SVM, the objective function has a regularization term c that balances the trade off between maximizing the margin and allowing some misclassifications.

- Decision Function
  The decision function of an SVM is

  $$f(x) = sign(w * x + b)$$

  where w is the weight vector, x is the input vector, b is the bias term and sign(*) is the sign function. The decision function classifies an input into one of the two classes based on the sign(positive or negative) of the expression.

- Support Vectors
  Support vectors are the items of data that lie on the margin or are misclassified, these data points are the most important for finding the optimal hyperplane and defining the margin.

- Parameter C
  The parameter C in SVM is used for regularization and control the trade-off between a

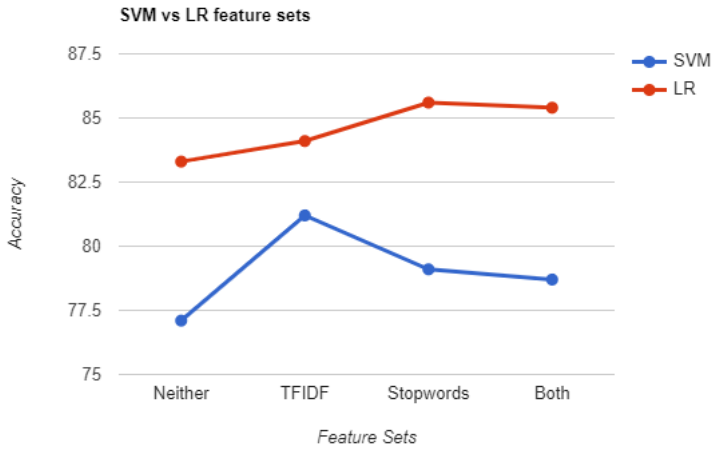
wider margin and allowing misclassifications. A smaller C makes for a wider margin and more misclassifications, while a larger C does the opposite.

- Kernel Parameters
  Different kernels, like the polynomial and RBF kernels, have additional parameters, degree and gamma respectively, that influence the shape of the boundary.

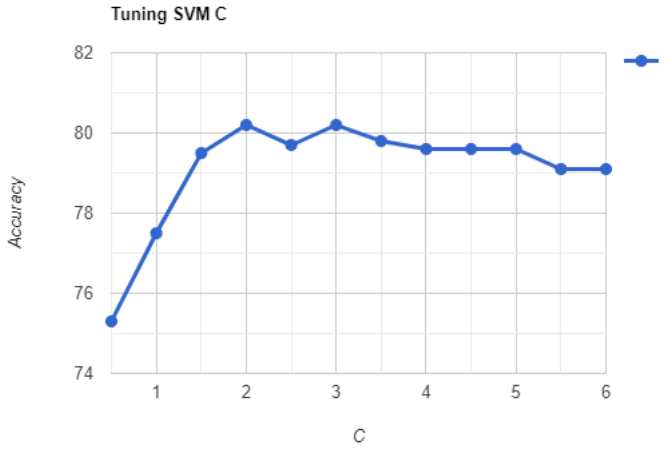
In the code snippet below is an implementation of SK-learns SVM classifier:

```
class SparseLinearSVCClassifier:
    def __init__(self, randomState = None):
        self.randomState = randomState
        self.model = LinearSVC(random_state = randomState)

    def train(self, texts, labels):
        vectorizer = CountVectorizer()
        X = vectorizer.fit_transform(texts)

        xTrain, xTest, yTrain, yTest = train_test_split(X, labels,
            test_size = 0.2, random_state = self.randomState)

        self.model.fit(xTrain, yTrain)

        yPred = self.model.predict(xTest)

        accuracy = accuracy_score(yTest, yPred)
        return accuracy

    def predict(self, texts):
        return self.model.predict(CountVectorizer.transform(texts))
```

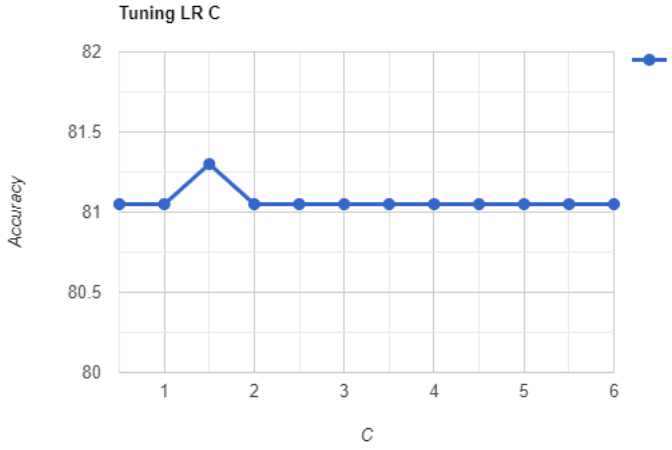
3.5.3 Hyperparameter optimization



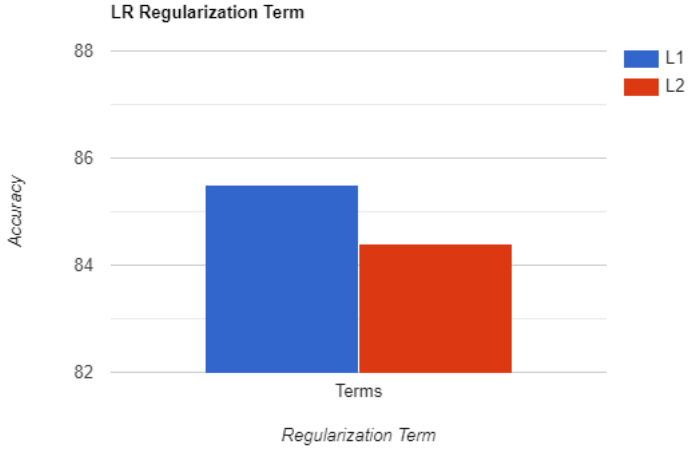
Based on the results of the experiments it is clear that LR mode outperforms the SVM consistently performing the best when TFIDF is not used and stopwords are removed.



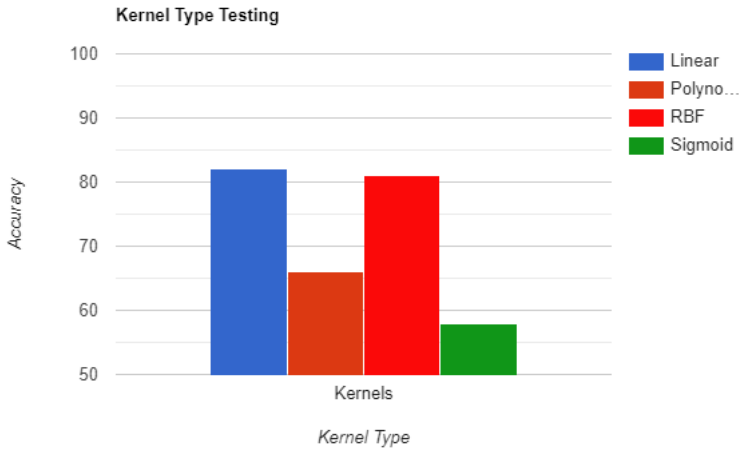
As mentioned earlier C is a hyperparameter in SVMs and to test, a grid search was performed. The result was that c values 2 and 3 outperformed others, however, since 2 was computationally quicker that will be used to proceed. It is worth mentioning that the default C is 1, changing this value greatly increased the accuracy of the model.



C is also a hyperparameter in Logistic Regression and the same method was used here, with a clear best at c = 1.5.



Different Regularization terms were also tested for Logistic Regression and it was found that, although slightly, L1 outperformed L2



Lastly, experiments were carried out to identify which SVM Kernel to use and there was a marginal difference between RBF(the default kernel) and the Linear kernel. The linear kernel outperforming the nonlinear ones without in depth hyperparameter tuning suggests that should we tune the non-linear kernels to achieve better results it would still be learning a linear function just with a non-linear prior. This also explains why logistic regression is outperforming SVMs as Logistic Regression mainly uses linear boundaries.

3.6 Bidirectional Encoder Representations from Transformers (BERT)

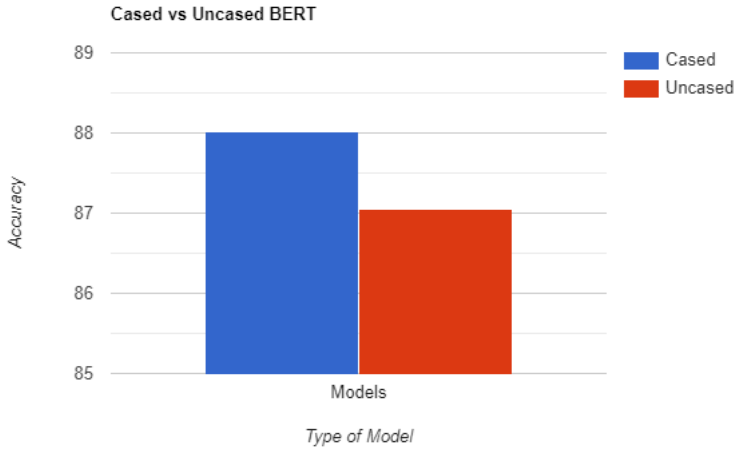
BERT is a pretrained NLP model developed by google. It is designed to capture context and relationships in language by pretraining on large amount of data. BERT is built on a transformer architecture, transformers allow for parallelization, attention mechanisms, and capturing long range dependencies in data. Unlike other language models, BERT is bi-directional allowing it to consider both the left and right context of each word, enabling a deeper understanding of context and semantics. BERT is trained on large corpora of text data using unsupervised learning to allow for larger test datasets not limited by labels. The model learns by predicting missing words in a sentence and understand the relationship between different sentences(next sentence prediction objective). BERT generates contextualized word embeddings, this means that a words representation relies on its context in a sentence, this is different from other word embeddings like Word2Vec that generate fixed embedding regardless of context. After it is pretrained BERT can be then fine tuned to perform specific tasks, such as text classification. The fine tuning process involves adapting the pretrained model to the nuances of a specific task using smaller, more targeted datasetsAlaparthi and Mishra (2020). The cased and uncased versions of the model refer to the letter case used in the pre-training language. In cased, text is kept in its original case whereas in uncased it is all converted to lowercase. Cased models are more suitable when the case of the words is relevant to the task at hand, such as named entity recognition, Uncased models are more often used in tasks where the case is irrelevant to the task, such as sentiment analysis and text classification. For the purpose of this assignment both versions of BERT were tested.

Below is a code snipped of the implementation of both cased and uncased BERT:

```
class CasedBertClassifier:
    def __init__(self, numLabels = 2, lr = 2e-5, epochs = 3):
        self.tokenizer =
            BertTokenizer.from_pretrained('bert-base-cased')
        self.model =
            BertForSequenceClassification.from_pretrained('bert-base-cased'
            , num_labels = numLabels)
        self.lr = lr
        self.epochs = epochs

    def tokenize_and_encode(self, texts, labels):
        tokens = self.tokenizer(texts, padding = True, truncation =
            True, return_tensors = 'pt')
        labels = torch.tensor(labels)
        return tokens, labels

    def train(self, xTrain, yTrain, xTest, yTest):
        tokenizedxTrain, yTrain = self.tokenize_and_encode(xTrain,
            yTrain)
        tokenizedxTest, yTest = self.tokenize_and_encode(xTest,
            yTest)

        trainData = TensorDataset(tokenizedxTrain['input_ids'],
            yTrain)
        testData = TensorDataset(tokenizedxTest['input_ids'], yTest)

        trainLoader = DataLoader(trainData, batch_size = 4, shuffle
            = True)
        testLoader = DataLoader(testData, batch_size = 4, shuffle =
            False)

        optimizer = AdamW(self.model.parameters(), lr = self.lr)

        for epoch in range(self.epochs):
            self.model.train()
            for batch in trainLoader:
                inputs, labels = batch
                outputs = self.model(inputs, labels=labels)
                loss = outputs.loss
                loss.backward()
                optimizer.step()
                optimizer.zero_grad()

            self.model.eval()
            testPreds = []
            testTrue = []
            with torch.no_grad():
                for batch in testLoader:
                    inputs, labels = batch
```

```
                outputs = self.model(inputs)
                logits = outputs.logits
                preds = torch.argmax(logits, dim=1)
                testPreds.extend(preds.numpy())
                testTrue.extend(labels.numpy())

        accuracy = accuracy_score(testTrue, testPreds)
        print(f"Cased BERT - Epoch {epoch + 1}/{self.num_epochs}, Validation Accuracy: {accuracy}")

        def predict(self,texts):
            tokenizedTexts, _ = self.tokenize_and_encode(texts, labels)
            self.model.eval()

            with torch.no_grad():
                outputs = self.model(**tokenizedTexts)
                logits = outputs.logits
                predictions = torch.argmax(logits, dim = 1).numpy()

            return predictions

class UncasedBertClassifier:
    def __init__(self, numLabels = 2, lr = 2e-5, epochs = 3):
        self.tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
        self.model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels = numLabels)
        self.lr = lr
        self.epochs = epochs

    def tokenize_and_encode(self, texts, labels):
        tokens = self.tokenizer(texts, padding = True, truncation = True, return_tensors = 'pt')
        labels = torch.tensor(labels)
        return tokens, labels

    def train(self, xTrain, yTrain, xTest, yTest):
        tokenizedxTrain, yTrain = self.tokenize_and_encode(xTrain, yTrain)
        tokenizedxTest, yTest = self.tokenize_and_encode(xTest, yTest)

        trainData = TensorDataset(tokenizedxTrain['input_ids'], yTrain)
        testData = TensorDataset(tokenizedxTest['input_ids'], yTest)

        trainLoader = DataLoader(trainData, batch_size = 4, shuffle
```

```
            = True)
        testLoader = DataLoader(testData, batch_size = 4, shuffle =
            False)

        optimizer = AdamW(self.model.parameters(), lr = self.lr)

        for epoch in range(self.epochs):
            self.model.train()
            for batch in trainLoader:
                inputs, labels = batch
                outputs = self.model(inputs, labels=labels)
                loss = outputs.loss
                loss.backward()
                optimizer.step()
                optimizer.zero_grad()

            self.model.eval()
            testPreds = []
            testTrue = []
            with torch.no_grad():
                for batch in testLoader:
                    inputs, labels = batch
                    outputs = self.model(inputs)
                    logits = outputs.logits
                    preds = torch.argmax(logits, dim=1)
                    testPreds.extend(preds.numpy())
                    testTrue.extend(labels.numpy())

            accuracy = accuracy_score(testTrue, testPreds)
            print(f"Uncased␣BERT␣-␣Epoch␣{epoch␣+␣
                1}/{self.num_epochs},␣Validation␣Accuracy:␣
                {accuracy}")

    def predict(self,texts):
        tokenizedTexts, _ = self.tokenize_and_encode(texts, labels =
            None)
        self.model.eval()

        with torch.no_grad():
            outputs = self.model(**tokenizedTexts)
            logits = outputs.logits
            predictions = torch.argmax(logits, dim = 1).numpy()

        return predictions
```

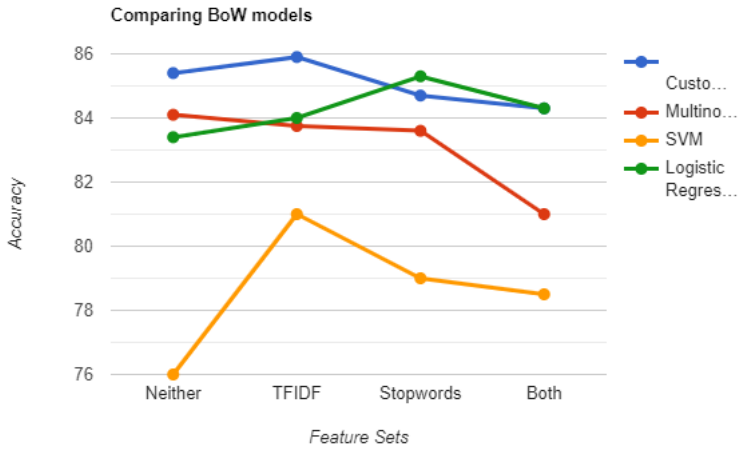


Based on the experiments conducted between cased and uncased BERT it is clear that the cased model performs better. This can be related to the loss of contextual information when converted to lower case. When capital letters are used at the start of sentences BERT can use embeddings to help distinguish between nouns or just words at the start of a sentence.

Chapter 4

Discussion

Collecting and combining the graphs from the experiments in the previous section can help visualize which combination of feature sets with which algorithm would be the best.



Above is a graph detailing the performance of the models across the different feature sets. The graph shows that only TFIDF and only Stopwords are the best performers. The following graph will enhance the difference between them by averaging out the performance of each feature set across the models.

This graph confirms that TDIDF alone is the best performer for this set of data. This also justifies the increased complexity of including TFIDF rather than simply removing the stopwords, however, that may be the better option given a different task or dataset. TFIDF and the removal of stopwords does not work well together as because when the stop words are removed the TFIDF algorithm will assign a lower score to more valuable words, since the words that normally get the low score have been removed. It also is unsurprising that using neither TFIDF and the removal of stopwords would perform the worst.

Taking into account all the previous experiments, the best performing feature sets and hyperparameters were used on the test set to evaluate and compare all the models mentioned against each other.

Model	Accuracy
Custom NB	88.864161846
BERT	88.351613138
SVM	87.715618486
Logistic Regression	86.841615466
MultinomialNB	83.486684186

As expected BERT performed the best of any of the built-in machine learning implementations, this is due to it using what is currently considered one of the best methods in natural language processing which is an attention based transformer. The custom implementation of Naive Bayes performed better than expected, but this can be attributed to hours of manual tuning and potentially some overfitting. To get the CustomNB working takes many more hours of deliberate coding which makes BERT better for more cases when the data sample is much larger than just 4000 reviews and cannot be retrained multiple times within a reasonable time limit. This also emphasizes the superiority of BERT out of the box over many general NLP tasks. The in-built version of MultinomialNB did not perform as good as expected, however, this can be due to the lack of optimization and tuning done. In comparison to SVM, linear regression has limited hyperparameter tuning. As a result, before hyperparameter optimization logistic regression outperformed SVM, but once the c value and kernel selection were done SVM outperformed logistic regression.

Chapter 5

Conclusions and Future Work

To conclude, BERT was the best performing model as it required the least effort and yielded impressive and consistent results across multiple seeds and cross validation. The fact that it is pre-trained on such a large corpus of text means that it can use the knowledge from different sources and apply them to the specific case after fine tuning. The one downside of BERT is that it takes a long time to train due to its nature of being a deep neural network with such a large number of parameters. There are many different ways to potentially improve and mitigate these issues, this can be experimenting with the different sized BERTs, trying different regularization techniques like dropout, adding learning rate schedules, and gradient clipping. All of which have been proven to benefit machine learning models in the past.

Bibliography

Alaparthi, S. and Mishra, M., 2020. Bidirectional encoder representations from transformers (bert): A sentiment analysis odyssey. 2007.01127.

Appel, O., Chiclana, F., Carter, J. and Fujita, H., 2016. A hybrid approach to the sentiment analysis problem at the sentence level. *Knowledge-based systems* [Online], 108, pp.110–124. New Avenues in Knowledge Bases for Natural Language Processing. Available from: https://doi.org/https://doi.org/10.1016/j.knosys.2016.05.040.

Barlow, H., 1989. Unsupervised Learning. *Neural computation* [Online], 1(3), pp.295–311. https://direct.mit.edu/neco/article-pdf/1/3/295/811863/neco.1989.1.3.295.pdf, Available from: https://doi.org/10.1162/neco.1989.1.3.295.

Cunningham, P., Cord, M. and Delany, S.J., 2008. *Supervised learning* [Online], Berlin, Heidelberg: Springer Berlin Heidelberg, pp.21–49. Available from: https://doi.org/10.1007/978-3-540-75171-7_2.

Devlin, J., Chang, M.W., Lee, K. and Toutanova, K., 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. 1810.04805.

Medhat, W., Hassan, A. and Korashy, H., 2014. Sentiment analysis algorithms and applications: A survey. *Ain shams engineering journal* [Online], 5(4), pp.1093–1113. Available from: https://doi.org/https://doi.org/10.1016/j.asej.2014.04.011.

Nick, T.G. and Campbell, K.M., 2007. *Logistic regression* [Online], Totowa, NJ: Humana Press, pp.273–301. Available from: https://doi.org/10.1007/978-1-59745-530-5_14.

Ouyang, X., Zhou, P., Li, C.H. and Liu, L., 2015. Sentiment analysis using convolutional neural network [Online]. *2015 ieee international conference on computer and information technology; ubiquitous computing and communications; dependable, autonomic and secure computing; pervasive intelligence and computing*. pp.2359–2364. Available from: https://doi.org/10.1109/CIT/IUCC/DASC/PICOM.2015.349.

Prasetijo, A.B., Isnanto, R.R., Eridani, D., Soetrisno, Y.A.A., Arfan, M. and Sofwan, A., 2017. Hoax detection system on indonesian news sites based on text classification using svm and sgd [Online]. *2017 4th international conference on information technology, computer, and electrical engineering (icitacee)*. pp.45–49. Available from: https://doi.org/10.1109/ICITACEE.2017.8257673.

Ramadhani, R.A., Indriani, F. and Nugrahadi, D.T., 2016. Comparison of naive bayes smoothing methods for twitter sentiment analysis [Online]. *2016 international conference on advanced computer science and information systems (icacsis)*. pp.287–292. Available from: https://doi.org/10.1109/ICACSIS.2016.7872720.

Salehinejad, H., Sankar, S., Barfett, J., Colak, E. and Valaee, S., 2018. Recent advances in recurrent neural networks. 1801.01078.

Yi, J., Nasukawa, T., Bunescu, R. and Niblack, W., 2003. Sentiment analyzer: extracting sentiments about a given topic using natural language processing techniques [Online]. *Third ieee international conference on data mining*. pp.427–434. Available from: https://doi.org/10.1109/ICDM.2003.1250949.