NLP Coursework Part A: Sentiment Analysis

Bachelor of Computer Science with Artificial Intelligence
The University of Bath
December 11, 2023

Contents

Abstract

Sentiment analysis poses a useful and technically challenging task in the field of Natural Language Processing. In this report we investigate the leading methods of sentiment analysis using a dataset of 4000 IMDB movie reviews, Maas et al. (2011), sorted into positive and negative classes based on their numerical rating. We aim to evaluate different techniques of feature extraction as well as

differing model architectures in order to determine the highest performing approach. We develop our own classification models for comparison with the existing sklearn implementations and compare the metrics of each (accuracy, precision, recall and f1 score). Through our evaluations we hope to gain further insight into the available methods of sentiment analysis and the factors which may increase or decrease performance of a given model and feature set. We aim to identify the challenges and associated solutions which may arise in our experimentations in order to develop a clearer understanding of currently available methodologies.

Chapter 1

Introduction

1.1 Sentiment Analysis

We can define sentiment analysis as the task of determining the sentiment latent in a given text, where sentiment is a positive or negative evaluation expressed through language, Taboada (2016). It can therefore be treated, as in our case, as a binary classification task. The task is important in the field of Natural Language Processing (NLP) due to its many applications and uses. Surveys (e.g. questionnaires) offer valuable qualitative data for businesses, however analysing responses is often time consuming and labour intensive. Sentiment analysis can be used in such cases to automate the task of reading through customer feedback, saving time and money for the surveyors. Similarly in the context of research, interviews produce high quality data with the drawback being that a researcher is required to transcribe and conduct a content analysis after the fact. Once again sentiment analysis has a clear application here, allowing researchers to automate the process and direct their resources elsewhere. Sentiment analysis thus demonstrates its importance clearly in the context of research and business. Coupled with the challenges required to overcome in order to conduct sentiment analysis, the task proves to be valuable in the context of NLP research.

1.2 Challenges

Sentiment analysis is by no means a trivial task and it poses many challenges to overcome during development. There are common issues faced when dealing with any Natural Language Processing (NLP) task as well as problems specific to sentiment analysis. Zipf's law states that the most commonly used word in a language appears twice as often as the second most common word, Upton and Cook (2014). The usefulness of these high frequency words in sentiment analysis is very limited as they appear ubiquitously across all classes. In the English language for example a frequently used connective such as "and" is of little use to us in understanding whether a given text is positive or negative in expression as it appears frequently across both classes. These words are referred to as stop-words and are often removed from the data altogether in NLP tasks. Specific to sentiment analysis, the use of language in certain contexts pose a difficult challenge to overcome. Sarcasm, idioms and expressions are challenging to identify in writing and often times an uncommon use of a word will confuse a model's analysis. Take the toy example sentence of a negative movie review below.

"This film cheesed me off, I would love to give the director a piece of my mind"

The uncommon expression "cheesed me off" (i.e. "annoyed me") has an unintuitive meaning and is particularly difficult for a language model to decode due to the novel use of common words (the noun cheese is used here as a verb). The use of the word "love" is overwhelmingly positive in most use cases, however in our example it is being used in conjunction with the expression "to give [object] a piece of my mind" to denote a negative feeling. In this example sentiment must be heavily inferred from context and knowledge of English expressions. The inference, contextual and linguistic knowledge required to decode such a statement poses a severe challenge in sentiment analysis whilst it may come naturally to a native English speaker.

1.3 Approaches

For feature selection and normalisation the following approaches are available to us

- N-grams
- Tokenization
- Stemming
- Lemmatization
- Stopword removal
- Term Frequency Inverse Document Frequency (TF-IDF)
- Frequency Normalisation
- Positive Pointwise Mutual Information (PPMI)

Of these approaches we intend to evaluate all but Frequency Normalisation and PPMI which have been omitted due to their complexity to implement. The following are the model architectures available to us

- Multinomial Naïve Bayes (MNB)
- Gaussian Naïve Bayes (GNB)
- Logistic Regression with Stochastic Gradient Descent (SGD)
- Support Vector Machine (SVM)
- Bidirectional Encoder Representatons from Transformers (BERT)

We will test and evaluate at least one version of each of the above models. We also intend to implement our own MNB and GNB models and compare these to the sklearn models, Pedregosa et al. (2011). Note that all models will be trained upon the same corpus, split into the same test, train and development sets. We will also evaluate the sklearn SVM and SGD models and the pre-trained BERT model from HuggingFace, Wolf et al. (2019). It must also be noted that the Guassian Naïve Bayes approach is not listed in the specification of this project. This different approach was chosen due to the use of TF-IDF feature normalisation which generates continuous non binary features in the range of [0, 1], thus prompting experimentation with a probability distribution based model such as GNB.

1.4 Related Work

In our experiments we use a subset of the IMDB movie review dataset, Maas et al. (2011), containing 2000 positive and 2000 negative movie reviews. As well as the methods we intend to test and evaluate there exist other approaches to sentiment analysis which are worthy of discussion. Deep learning techniques such as Recurrent Neural Networks (RNNs) and Long Short Term Memory (LSTM) networks can consider historical contextual data within a text with more nuance than any of the models we intend to test, Yu et al. (2019). The potentially higher performance of these models is offset by their computational complexity however as these approaches are by no means trivial to implement and can potentially consume a large amount of computational resources during training.

On the other end of the spectrum dictionary based approaches are perhaps the most simple methods of sentiment analysis, Van Atteveldt, Van der Velden and Boukes (2021). Scores are assigned to texts through use of predefined dictionaries which score words on their sentiment (positive, negative, or neutral). These simpler methods may lack the complexity to generalise well however and will be unable to consider context in their evaluation. The machine learning models we intend to test all fall under the umbrella of supervised learning approaches and were we to lack labelled data we would require unsupervised approaches instead. Latent Dirichlet Allocation (LDA) Blei, Ng and Jordan (2003), considers documents as mixtures of various topics and topics as distributions of words in order to generate a class label for a given text. Were we dealing with unsupervised approaches and more class labels than binary (positive, negative), LDA would be a sensible approach to take.

Chapter 2

Experiments and Results

2.1 N-Grams

N-Gram generation is an NLP technique which we can use to extract features from our dataset in order to conduct sentiment analysis. With N-Gram generation we group together consecutive *n* runs of words and use these as features. The intuition behind N-Grams is that, to borrow from linguist John Firth, "you shall know a word by the company it keeps". Contextual information is highly important in NLP tasks as the meaning of a given word can significantly vary depending on its context. Our model may thus achieve higher performance when extracting consecutive pairs (bigrams) or triplets (trigrams) of words from the corpus as opposed to single tokens (unigrams) as contextual information is better preserved in the former. N-Gram generation with higher values of *n* does increase the computational complexity substantially however as we generate a higher number of unique features. Below is an example of N-Gram generation (*for n = 1, 2, 3*) using a quote from semiotician Jacques Derrida.

"There is nothing outside the context", Derrida (2016)

```
Unigrams: (There) (is) (nothing) (outside) (the) (context)
Bigrams: (There is) (is nothing) (nothing outside) (outside the) (the
   context)
Trigrams: (There is nothing) (is nothing outside) (nothing outside the)
    (outside the context)
```

Despite the advantage of increased contextual information when using N-Grams, we use unigram feature generation for our feature sets. After preliminary experiments it becomes evident that the added computational complexity arising from a higher feature count renders bigrams and trigrams as unviable options for feature extraction. Table 2.1 shows the number of unique words in the shared vocabulary of the corpus, accuracy achieved with our own MNB model (*see section 2.6*) and time taken to train across varying N-Gram sizes.

N-Grams	Words in Shared Vocabulary	Training Time (minutes)	Accuracy
Unigrams	44377	1	0.852
Bigrams	392191	92	0.6375
Trigrams	509607	Untested	Untested

Table 2.1: Evaluation of N-Grams using our own MNB model (*see section 2.6*)

The higher number of unique words for bigrams and trigrams renders them unusable as the time complexity of training increases massively. As shown in Table 2.1 the time taken for training a feature set using bigrams is far higher than the same set using unigrams. In addition the accuracy of the bigram feature set is significantly lower than that of the unigram set. There could be multiple reasons for this, such as a lack of quality features in the over inflated bigram shared vocabulary. Consequently we use unigrams in our feature sets, as it is unviable to train every model for over an hour each time during development and because the unigram set appears to afford us a higher accuracy. Trigrams were left untested due to lack of computational resources. Potential further experimentation could be conducted using bigrams and trigrams given greater computational resources, however we will focus solely on unigrams for our investigation into sentiment analysis.

2.2 Feature Selection

Through the use of a combination of Tokenization, Stemming/Lemmatization, Stop-word Removal, and TF-IDF scoring we generate three feature sets to train and evaluate our models upon. Below is a short description of each technique and its application to sentiment analysis.

Tokenization: Tokenization is simply the process of converting a block of text into individual tokens, which are often single words. In doing so we complete the first step in the feature selection process as individual tokens can be used as features whereas entire paragraphs (in most cases) cannot. For our purposes we will split our texts into tokens whereby each word is a token and whitespace, punctuation and stop-words (*see below*) are removed.

Stop-word Removal: In accordance with Zipf's law, Upton and Cook (2014), the most commonly used words in a language are seen so often across the corpus that their usefulnessin sentiment analysis becomes limited. These words are present in both positive and negative examples with similar frequencies, meaning they cannot be used to discern between classes. Conversely words which appear rarely may be of great use to us in categorising a review as positive or negative. For example if we know that the word "dreadful" appears in ten reviews, nine of which are negative, the word then becomes a useful feature for discerning negative reviews as opposed to the word "and" which appears equally as frequently in both classes. Therefore we simply remove these frequently used useless words, which are referred to as stop-words, in order to preserve only those features which are useful for our model in distinguishing between classes.

Stemming/Lemmatization: One challenge faced in NLP tasks is the notion of morphology within language. Morphology refers to conjugations, prefixes and suffixes applied to words to change their meaning. For example the word run can become runner, running, ran etc. For sentiment analysis we would like to simplify these morphed words in order to glean the common meaning for our model (in this case that would be "run"). Stemming and lemmatization allow us to do this by reducing words to either their stem or lemma. A word stem is the root of a word, Merriam-Webster (1990), and a lemma is "a word without any morphological changes made to it", Merriam-Webster (1990). The meaning of the words in our corpus is thus preserved whilst different forms of the same word are removed. By reducing the words in our dataset to their lemmas or stems we reduce the complexity of the task and allow for simpler feature selection in sentiment analysis.

TF-IDF: Similar to some of the aforementioned methods the motivation behind Term Frequency - Inverse Document Frequency (TF-IDF) scoring is to utilise word frequency data to extract

features from our dataset. Term Frequency refers to the count of terms within a single document as a proportion of the total words in said document. Inverse Document Frequency is the reciprocal of the number of documents containing a given word as a proportion of the total number of documents. We then multiply the two values together to generate a TF-IDF score which is then used as a feature in our classification models. A high TF-IDF score for a given feature indicates that it is of use as a distinguishing feature between classes due to its rare occurence in the corpus. For example a word such as "and" will have a low TF-IDF score due to its abundance across documents of all classes. TF-IDF scores also allow us a method of feature normalisation as the original text consisting of words has now been encoded as numerical values.

2.3 Feature Sets

Having discussed the feature selection methods available we now list the combinations of approaches used in generating our three feature sets. Note that all sets have been subject to tokenisation.

1. **Stemming**: Stemming with Unigrams and TF-IDF. As explained in section 2.1 any N-Gram higher than unigrams is unviable due to lack of computational resources. In this set we choose not to remove stop-words in order to assess the performance of this feature set against those which use stop-word removal, in order to discern its usefulness in sentiment analysis.
2. **Stemming + Stop-words**: Stemming with Unigrams, stop-word removal and TF-IDF. In this set we remove stop-words to compare with **Set 1** as both sets are otherwise identical.
3. **Lemmatization + Stop-words**: Lemmatization with Unigrams, stop-word removal and TF-IDF. This set is identical to **Set 2** aside from the use of lemmatization over stemming, allowing us to compare the performance of the two methods.

2.4 Data Splits

For our sentiment analysis models we split the dataset into three subsets; training (70%), testing (20%) and development (10%). The training set is used to calculate the relevant likelihoods, priors and weights of our models. The test set is used to evaluate the performance of our models after hyperparameter tuning. Finally the development set is used after training and before testing in order to tune the relevant hyperparameters of our models and make any necessary adjustments in order to increase performance. We use the sklearn *test_train_split* function with a constant value for the *random_state* parameter in order to ensure the same splits across all models. Defining a sensible split of the data is crucial in implementing a model that generalises well. Separate sets must be reserved for testing and development because the model will achieve 100% accuracy if evaluated on the training data. Similarly we must separate the development and testing data in order to fairly assess the performance of our models, as test data is by definition unseen. By defining a split of data in this way we aim to maximise the performance of our models and increase their generalisability.

2.5 Bayesian Models

Bayesian models are a popular choice for classification tasks and in this case they will be used for sentiment analysis. The intuition behind Bayesian models stems from Bayes' rule which states that the probability of an event A occurring given event B can be calculated when using the Naïve independence assumption.

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

In training we calculate the priors and likelihoods for our classes (positive and negative) which are then used to classify new inputs. Priors refer to the proportion of documents belonging to a given class.

$$P(C) = \frac{\text{Number of documents belonging to class } C}{\text{Total number of documents}}$$

Likelihoods are calculated per feature and indicate the probability that a document with said feature belongs to a given class.

$$P(\text{feature } f | \text{ class } C) = \frac{\text{Number of documents in class } C \text{ with feature } f}{\text{Total number of documents in class } C}$$

Putting the above equations together we can estimate the probability that a document x belongs to a given class C. Because the denominator *p(x)* remains constant across all calculations it is dropped, as it has no effect on the final comparison of probabilities between classes. Thus we have a formula for determining the probability that a given document belongs to a given class which will be the central mechanism of our Bayesian models.

$$P(C|x) = \prod_{i=1}^{n} P(x_i|C) \cdot P(C)$$

We often take the logarithm of all calculations in order to avoid dealing with very small numbers, giving us our final mathematical model.

$$P(C|x) = \sum_{i=1}^{n} \log P(x_i|C) + \log P(C)$$

2.6 Multinomial Naïve Bayes

For evaluating Multinomial Naïve Bayes (MNB) for sentiment analysis on our dataset we implement our own model and compare it against the sklearn implementation. We make the Naïve assumption that our features are independent of one another, i.e. that the presence of one feature does not impact the likelihood of another feature's presence. This assumption is made purely for modelling purposes, in actuality it does not stand. This particular approach to Bayesian learning is known as Multinomial as we assume a multinomial distribution of features among our documents, from which we can infer class likelihoods and probabilities. The class likelihoods and priors are calculated in training and used to classify new documents as per the formulae in section 2.5. A vector of TF-IDF scores is passed into the model as input when classifying a new document. We then sum the logarithms of the feature likelihoods for each present feature for each class (positive and negative). After then adding the logarithm of the

given class prior we have a value for each class representing the probability of the input document belonging to said class. In order to avoid taking the logarithm of 0 at any point we apply Laplace Smoothing to the likelihoods, whereby we add a constant *alpha* (= 1) to avoid any 0 values. Whichever value is higher determines the class label. Note that the values being compared are not strictly probabilities in the range of [0, 1] due to the application of the logarithm in order to avoid very small probabilities. Table 2.2 shows an evaluation on dev set of both models.

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
Stemming	Ours	0.802	0.927	0.623	0.745
	Sklearn	0.835	0.779	0.901	0.836
Stemming + Stop-Words	Ours	0.875	0.927	0.794	0.855
	Sklearn	0.856	0.816	0.892	0.852
Lemmatization + Stop-Words	Ours	0.867	0.939	0.762	0.841
	Sklearn	0.865	0.846	0.865	0.856

Table 2.2: Comparison of performance on feature sets for our own Multinomial Naïve Bayes model and the pre-built sklearn MNB model.

It is clear that removing stopwords increases performance across both models. Lemmatization and stemming with stop-word removal both perform highly on the development set with little difference in evaluation metrics. We will evaluate our models on the test set using stemming and stop-word removal because of the slightly higher accuracy in comparison to the lemmatization set. Our own implementation of MNB generally matches or outperforms the sklearn model on the feature sets with stop-word removal. Table 3.1 (*see section 3.1*) shows the performance on the test set of both models on the stemming with stop-word removal feature set.

2.7 Gaussian Naïve Bayes

As opposed to MNB, Guassian Naïve Bayes (GNB) assumes a Guassian distribution of values for each feature. In practice this means that we calculate a probability distribution for each possible feature during training and fit the value for a feature in the given document to the distribution during classification. As before we implement our own GNB model and compare its performance to the pre-built sklearn model. GNB has the added potential advantage over MNB of being more robust and less sensitive to small datasets, as well as better suited to continuous data. In our own implementation of GNB we pre-process the mean and variance of each feature's TF-IDF scores across all documents. This is then used to calculate the likelihood of a particular document's feature belonging to a given class using the following formula.

$$P(x_i|C) = \frac{1}{\sqrt{2\pi\sigma_{C,i}^2}} \cdot \exp\left(-\frac{(x_i - \mu_{C,i})^2}{2\sigma_{C,i}^2}\right)$$

We take the logarithm of this class likelihood and sum it for each feature in the input document and finally we add the logarithm of the class prior, giving us a value indicating the probability of the document belonging to each class as before with our MNB model. Table 2.4 shows the results on the development set of our own GNB model and the prebuilt sklearn model across our three feature sets.

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
Stemming	Ours	0.860	0.831	0.879	0.854
	Sklearn	0.650	0.612	0.672	0.641
Stemming + Stop-Words	Ours	0.865	0.829	0.892	0.860
	Sklearn	0.650	0.611	0.677	0.643
Lemmatization + Stop-Words	Ours	0.867	0.833	0.892	0.861
	Sklearn	0.635	0.593	0.686	0.636

Table 2.3: Comparison of performance on feature sets for our own Gaussian Naïve Bayes model and the pre-built sklearn GNB model.

Once again the feature sets with stop-word removal outperform the set without and stemming and lemmatization with stop-word removal perform very similarly. As opposed to our MNB results our model decisively outperforms the sklearn implementation, achieving much higher scores for every metric across all sets. For evaluation on our test set we will once again use stemming with stop-word removal, as the performance of sklearn's model is higher in this set than with lemmatization. Table 3.2 (*see section 3.2*) shows the performance of both models on the test set.

2.8 Logistic Regression

With Logistic Regression we train the weight vector w of a model such that when multiplied by a feature vector as input and passed through a sigmoid function a class probability is output, Menard (2002). The sigmoid function is necessary to constrain the value between [0, 1] and thus output a probability. For Logistic Regression (and Support Vector Machines) we are required to encode our features as a one-hot vector meaning that our feature vector consists only of binary values. We choose to lose the information gained by TF-IDF scoring in order to utilise these one-hot vectors instead. Stochastic Gradient Descent (SGD) is an algorithm often used to update the weight vector w during training. A loss function is defined to measure the disparity between the results of the classifier and the true class labels. SGD iteratively minimises the loss function through a gradient descent technique which updates weights such that the loss function moves towards a local minimum Menard (2002). We use the sklearn *LogisticRegression* model in our evaluation and test the SGD algorithm in comparison to others such as the L-BFGS solver.

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
Stemming	Sklearn	0.838	0.811	0.848	0.823
Stemming + Stop-Words	Sklearn	0.833	0.812	0.834	0.823
Lemmatization + Stop-Words	Sklearn	0.858	0.841	0.857	0.849

Table 2.4: Comparison of performance on feature sets for the sklearn Logistic Regression model.

The best feature set is evidently lemmatization with stop-word removal. Continuing to work with the development set we will now optimise the hyperparameters of the sklearn LogisticRegression model. Following hyperparameter optimisation we evaluate the final tuned model on the test set. Below are the hyperparameters we experimented with during the fine tuning of the model, all of which were tested using the development set and lemmatization with stop-word removal and compared to the baseline sklearn model with no hyperparameter tuning using SGD as the update algorithm.

1. **Solver**: Specifies the algorithm used for optimisation. Experimenting with different solvers reveals that *lbfgs* achieves a higher performance on the development set over *sag* (SGD), despite the potential advantages of a stochastic gradient descent algorithm. Due to this higher performance we choose to use the lbfgs solver instead of the SGD algorithm in our final tuned model.
2. **Regularization Parameter (C)**: Defines the strength of regularization as inversely proportional to the value given. Hence a lower value causes higher regularization and vice versa. During development numeruous values for C were evaluated and ultimately a value of 1.5 was found to be optimal.
3. **Penalty**: Determines the norm of the penalty. For the lbfgs solver either *l2* or *None* can be specified in the penalty parameter. An L2 penalty norm gives a higher accuracy than no norm whatsoever, thus we use an L2 norm in our model.
4. **N_jobs**: Controls the number of CPU cores used when parallelizing. Setting this parameter to -1 indicates an unlimited number of cores. Surprisingly the task was performed faster when n_jobs = None, i.e. when the task was not parallelized. For this reason we use n_jobs = None in our model, constraining the task to a single CPU core.

2.9 Support Vector Machines

Support Vector Machines (SVMs) operate by aiming to find the optimal hyperplane separating different classes whilst maximising the margin between said classes, Yu et al. (2019). The Support Vectors are those data points lying closest to the decision boundary that the hyperplane aims to divide. As with Logistic Regression, SVMs take one-hot vectors as input. SVMs are an efficient method of classification due to their generalisation capabilities and ability to deal with high dimensionality data. Once again we utilise the sklearn library in order to evaluate the performance of an SVM model for sentiment analysis. Training an SVM model can be potentially time consuming as SVMs do not scale well with large datasets. In addition complexity increases as the number of features increases, meaning that SVMs may not be the optimal model choice were we to test with any N-Gram selection higher than unigrams.

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
Stemming	Sklearn	0.852	0.825	0.865	0.844
Stemming + Stop-Words	Sklearn	0.852	0.825	0.865	0.844
Lemmatization + Stop-Words	Sklearn	0.850	0.821	0.865	0.843

Table 2.5: Comparison of performance on feature sets for the sklearn Support Vector Machine model.

Interestingly our results show that the removal of stop-words has no effect on the results of the SVM classifier as we see identical results for stemming with and without stop-word removal. We will choose stemming with stop-word removal as the best feature set for use with SVMs due to its higher performance over stemming without stop-word removal in our other models. We now proceed to optimise the hyperparameters of the sklearn model and subsequently evaluate upon the test set (*see table Table 3.4*)

1. **Regularization Parameter**: Defines the strength of regularization as inversely proportional to the value given. Hence a lower value causes higher regularization and vice versa.

During development a value of 0.9 was found to outperform the default value of 1.0, hence we use this in our final model.

2. **Kernel**: Specifies the kernel to be used in the algorithm. Experiments showed that the *rbf* kernel achieved a much higher accuracy over the *linear* kernel, making it a suitable choice for our fine tuned model.

3. **Gamma**: Determines the kernel coefficient. Having tested the two options, *scale* and *auto*, it is clear that auto is the vastly superior choice, as scale achieves the lowest possible accuracy of around 0.50. The value of the coefficient using auto is defined as *1 / n features*.

2.10 BERT

Bidirectional Encoder Representations from Transformers (or BERT) is an NLP model that utilises a transformer architecture with deep learning techniques. The Bidirectional Representation element of the model refers to the fact that BERT uses a self-attention mechanism to consider both the left and right context of a masked word, where the model is attempting to predict the masked word as in a masked language model. This affords the model a deeper contextual understanding of a given text making BERT a potentially high performing solution for the task of sentiment analysis.

In our testing we evaluate both the cased and uncased versions of the pre-trained BERT model available from HuggingFace Wolf et al. (2019). Case here refers to the capitalisation of letters, as uncased BERT does not consider capitalisation whereas cased BERT does. Identifying proper nouns may therefore be less accurate in the uncased BERT model, potentially leading to lower performance in sentiment analysis. We test both versions of the model on the same test, train and development split as with all previous models. The BERT models have a much higher time complexity and require a much larger amount of computational resources than any other models thus far. This is due to the complexity of the model architecture itself as this is the first deep learning transformer and self-attention model to be evaluated. Due to the complexity requirements we have trained the BERT model using Google's Colab environment, Bisong et al. (2019), through which we can access a powerful GPU that can complete training in approximately 5 minutes. The performance of the BERT model could likely be increased given more training data, as a dataset of 4000 samples is insufficient for BERT to achieve its maximal performance. Table 2.10 shows the results of both the cased and uncased BERT models on the development set. Note that the usual feature sets do not apply in this case as BERT requires its own encoding and tokenization methods.

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
BERT Features	Uncased	0.846	0.831	0.839	0.835
BERT Features	Cased	0.844	0.803	0.879	0.839

Table 2.6: Comparison of performance on the development set by Google's cased and uncased BERT model.

Chapter 3

Discussion

3.1 Mutinomial Naïve Bayes Evaluation

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
Stemming + Stop-word Removal	Ours	0.809	0.876	0.729	0.796
	Sklearn	0.820	0.809	0.849	0.829

Table 3.1: Comparison on the test set of performance of stemming with stop-word removal feature set for our own Multinomial Naïve Bayes model and the pre-built sklearn MNB model.

Both our own implementation and the sklearn model perform well on the test set with both achieving accuracies higher than 80%. Sklearn's model slightly outperforms our own across all metrics aside from precision. This may explain the disparity in performance, as it is possible that our model's higher precision leads to a lower recall and thus a lower accuracy. We aimed to achieve the same or higher performance in comparison to the sklearn implementation however the disparity in evaluation metrics is minimal to the point that our model remains viable. Potential improvements to precision and recall could be made by examining the values assigned for each class in cases of ambiguous samples, i.e. in cases where our model has a low certainty about the given class label.

3.2 Gaussian Naïve Bayes Evaluation

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
Stemming and Stop-word Removal	Ours	0.828	0.811	0.866	0.837
	Sklearn	0.653	0.658	0.671	0.664

Table 3.2: Comparison on the test set of performance of stemming with stop-word removal feature set for our own Gaussian Naïve Bayes model and the pre-built sklearn GNB model.

For the GNB models our own implementation vastly outperforms the sklearn model on every available metric. Such a large disparity in accuracies can be explained through a number of differences between the two implementations of the method. Firstly, the sklearn GNB model has a much lower time complexity than our own model, with the former training in a matter of seconds as opposed to minutes for the latter. This could suggest an optimisation that sacrifices

accuracy for speed in the case of the sklearn model. In our implementation we use multiple slightly unconventional techniques to improve performance. The first is Laplace Smoothing, which is not regularly used in the context of a GNB model and is more commonly found in MNB implementations. We found that our model performed worse without Laplace Smoothing as it encountered 0 and infinity values during its calculations. Secondly we choose to sum the logarithms of our likelihoods and priors which is once again a technique seen more often in MNB models. We did this in order to avoid errors arising from calculations involving very small numbers. These differences in approach may help to explain the large performance gap between our own and the sklearn GNB model.

3.3 Logistic Regression Evaluation

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
Lemmatization + Stop-words	Sklearn	0.836	0.833	0.851	0.842

Table 3.3: Performance on the test set of the sklearn Logistic Regression model with hyperparameter optiisation.

For our final logistic regression model we utilise one hot vectors as features over TF-IDF scores as we found an increase in performance when using the former. During development we found that the sklearn lbfgs solver outperformed the SGD based sag solver for our purposes of sentiment analysis. We used the lbfgs solver in conjunction with other hyperparameter optimisations to fine tune the model and achieve the results as displayed in Table 3.3. Once again our results for this model are very close to the average performance of models thus far. There is room for further hyperparameter tuning and more extensive testing of the sklearn SVM model, however these experiments remain outside the scope of this project. The logistic regression model achieves the highest f1 score of all models which suggests a robust performance when dealing with unseen data.

3.4 Support Vector Machines Evaluation

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
Lemmatization + Stop-words	Sklearn	0.826	0.809	0.866	0.836

Table 3.4: Performance on the test set of the sklearn Support Vector Machine model with hyperparameter optimisation.

The sklearn SVM model achieves a similar accuracy to the highest performing MNB and GNB models tested thus far. Similar results across multiple models is expected when using the same features and split of the dataset and indicates to us that our own implementations are functioning properly. The SVM model achieves a high recall, from which we can infer that the model is effectively discerning the true positive instances of the dataset, suggesting that the hyperplane has been properly adjusted during training. It appears that an SVM approach is well suited to this task, as the sklearn SVM model achieves the highest overall performance when compared with the other models (aside from BERT). Once again further work could be conducted in tuning the hyperparameters given more time and resources. Given that all of our models achieve similar accuracies it would be of interest to examine the differences between models given a much larger dataset and more computational resources.

3.5 BERT Evaluation

Feature Set	Model	Accuracy	Precision	Recall	F1 Score
BERT Feature Set	Uncased	0.891	0.889	0.900	0.895
BERT Feature Set	Cased	0.883	0.866	0.912	0.888

Table 3.5: Performance on the test set of the Google's cased and uncased BERT models

Due to the similar performance from both BERT models on the development set we evaluated both on the test set in order to determine the best model. Surprisingly the uncased model outperforms the cased model on the test set, which could be due to the fact that the data has less variation in the uncased encodings. By taking into account the case of words, the cased model may encode more unique features, leading to higher dimensionality and thus lower performance in the case of a small dataset. The disparity in performance between the two models remains minimal however, and both BERT models appear to outperform all other tested models. This is expected due to their complex architectures and is reflected in time complexity and computational resources required.

Chapter 4

Conclusion

4.1 Conclusion

We have experimented with different approaches to feature analysis and modelling in the context of sentiment analysis using a dataset of 4000 IMDB movie reviews. Our results found that stop-word removal produces a marked increase in performance across all models tested. We found that any N-Gram size higher than unigrams was unviable in our circumstances due to time complexity and computational demand. We used TF-IDF scoring to normalise our features which appeared to positively affect performance.

Of the approaches tested we found that all models performed very similarly, with all bar one achieving accuracies of around 80%. We determine that the best performing model (not including BERT) is sklearn's SVM model and the worst is sklearn's GNB model. Our own implementation of MNB achieves a nearly equal performance to the sklearn implementation. For the GNB models our own implementation vastly outperforms the sklearn model, scoring significantly higher on every evaluation metric. We found that hyperparameter optimisation led to increased performance with our SVM and Logistic Regression models and would be interested in further optimisation given more time. Through our experimentation we found that our dataset is insufficiently large for optimised performance with BERT, however we still achieve high accuracies using the model. We note that the time complexity of the BERT model is significantly higher than any other.

4.2 Future Work

Given the size of our dataset it would be interesting to investigate the differences between sentiment analysis models on a very large dataset, for example the original IMDB dataset of size 50,00 Maas et al. (2011). For our logistic regression and SVM models the effects of hyperparameter tuning on performance could be examined further, with more extensive testing and more combinations experimented with. Finally further research could be conducted into feature extraction and normalisation techniques, as we experimented with only one tokenizer and chose not to implement PPMI for feature normalisation. Examining the effects of these approaches on model performance could form the basis of interesting future work in the field of sentiment analysis.

Bibliography

Bisong, E. et al., 2019. *Building machine learning and deep learning models on google cloud platform*. Springer.

Blei, D.M., Ng, A.Y. and Jordan, M.I., 2003. Latent dirichlet allocation. *Journal of machine learning research*, 3(Jan), pp.993–1022.

Derrida, J., 2016. *Of grammatology*. Jhu Press.

Maas, A., Daly, R.E., Pham, P.T., Huang, D., Ng, A.Y. and Potts, C., 2011. Learning word vectors for sentiment analysis. *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*. pp.142–150.

Menard, S., 2002. *Applied logistic regression analysis*, 106. Sage.

Merriam-Webster, I., 1990. *Webster's ninth new collegiate dictionary*, vol. 10. Merriam-Webster.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. et al., 2011. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct), pp.2825–2830.

Taboada, M., 2016. Sentiment analysis: An overview from linguistics. *Annual review of linguistics*, 2, pp.325–347.

Upton, G. and Cook, I., 2014. *A dictionary of statistics 3e*. Oxford quick reference.

Van Atteveldt, W., Velden, M.A. Van der and Boukes, M., 2021. The validity of sentiment analysis: Comparing manual annotation, crowd-coding, dictionary approaches, and machine learning algorithms. *Communication methods and measures*, 15(2), pp.121–140.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M. et al., 2019. Huggingface's transformers: State-of-the-art natural language processing. *arxiv preprint arxiv:1910.03771*.

Yu, Y., Si, X., Hu, C. and Zhang, J., 2019. A review of recurrent neural networks: Lstm cells and network architectures. *Neural computation*, 31(7), pp.1235–1270.

nlp_environment

December 12, 2023

Sentiment Analysis

2023 NLP Coursework Part A. First we must import the necessary packages

[ ]:
```
import nltk
import string
import numpy as np

nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')

from nltk.corpus import stopwords
from nltk.stem.lancaster import LancasterStemmer
from nltk.stem import WordNetLemmatizer

np.random.seed(42)
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\adame\AppData\Roaming\nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\adame\AppData\Roaming\nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\adame\AppData\Roaming\nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

Extracting the data

[ ]:
```
import os
def read_corpus(directory):
    files = [f for f in os.listdir(directory) if f.endswith('.txt')]
    corpus = []
    for file in files:
        with open(os.path.join(directory, file), 'r', encoding='utf-8') as f:
            document = f.read()
            corpus.append(document)
    return corpus
```

[ ]:
```
positive_corpus = read_corpus("data/pos/")
negative_corpus = read_corpus("data/neg/")
corpus = positive_corpus + negative_corpus
positive_labels = len(positive_corpus)
negative_labels = len(negative_corpus)
corpus_length = len(corpus)

#sanity check, should be the same every time
print(positive_corpus[0])
print(negative_corpus[1])
```

```
Homelessness (or Houselessness as George Carlin stated) has been an issue for
years but never a plan to help those on the street that were once considered
human who did everything from going to school, work, or vote for the matter.
Most people think of the homeless as just a lost cause while worrying about
things such as racism, the war on Iraq, pressuring kids to succeed, technology,
the elections, inflation, or worrying if they'll be next to end up on the
streets.<br /><br />But what if you were given a bet to live on the streets for
a month without the luxuries you once had from a home, the entertainment sets, a
bathroom, pictures on the wall, a computer, and everything you once treasure to
see what it's like to be homeless? That is Goddard Bolt's lesson.<br /><br />Mel
Brooks (who directs) who stars as Bolt plays a rich man who has everything in
the world until deciding to make a bet with a sissy rival (Jeffery Tambor) to
see if he can live in the streets for thirty days without the luxuries; if Bolt
succeeds, he can do what he wants with a future project of making more
buildings. The bet's on where Bolt is thrown on the street with a bracelet on
his leg to monitor his every move where he can't step off the sidewalk. He's
given the nickname Pepto by a vagrant after it's written on his forehead where
Bolt meets other characters including a woman by the name of Molly (Lesley Ann
Warren) an ex-dancer who got divorce before losing her home, and her pals Sailor
(Howard Morris) and Fumes (Teddy Wilson) who are already used to the streets.
They're survivors. Bolt isn't. He's not used to reaching mutual agreements like
he once did when being rich where it's fight or flight, kill or be killed.<br
/><br />While the love connection between Molly and Bolt wasn't necessary to
plot, I found "Life Stinks" to be one of Mel Brooks' observant films where prior
to being a comedy, it shows a tender side compared to his slapstick work such as
Blazing Saddles, Young Frankenstein, or Spaceballs for the matter, to show what
it's like having something valuable before losing it the next day or on the
other hand making a stupid bet like all rich people do when they don't know what
to do with their money. Maybe they should give it to the homeless instead of
using it like Monopoly money.<br /><br />Or maybe this film will inspire you to
help others.
This film is about a male escort getting involved in a murder investigation that
happened in the circle of powerful men's wives.<br /><br />I thought "The
Walker" would be thrilling and engaging, but I was so wrong. The pacing is
painfully and excruciatingly slow, that even after 40 minutes of the film
nothing happens much. Seriously, the first hour could be condensed into ten
minutes. That's how slow it is.<br /><br />The fact that it lacks any thrills or
```

action scenes aggravates the boredom. It's almost shocking that even argument scenes are so plain and devoid of emotion. Maybe it is because of the stiff upper lip of the higher social class?

It's sad that "The Walker" becomes such a boring mess, despite such a strong cast. Blame it on the poor plot and even worse pacing.

Feature Generation Using Ngrams

[ ]:
```
from nltk import word_tokenize
from nltk import ngrams

def text_to_ngrams(sentence, n, remove_stopwords=True):
    stoplist = set(stopwords.words('english')) #stop-words to remove
    if not remove_stopwords:
        stoplist = set()
    tokenised_words = [word for word in word_tokenize(sentence.lower()) if word␣
 ↪not in stoplist and word not in string.punctuation
                       and word != "br"]
                    #a list of tokenised words with stop-words, punctuation,␣
 ↪and <br>s removed
    zipped_grams = ngrams(tokenised_words, n) #apply nltk's ngrams algorithm
    return list(zipped_grams)
```

[ ]:
```
sentence = "I am Ozymandias, king of kings, look upon my works ye mighty and␣
 ↪despair"
grams = text_to_ngrams(sentence, 3)
print(grams)
for gram in grams:
    print(gram)
```

```
[('ozymandias', 'king', 'kings'), ('king', 'kings', 'look'), ('kings', 'look',
'upon'), ('look', 'upon', 'works'), ('upon', 'works', 'ye'), ('works', 'ye',
'mighty'), ('ye', 'mighty', 'despair')]
('ozymandias', 'king', 'kings')
('king', 'kings', 'look')
('kings', 'look', 'upon')
('look', 'upon', 'works')
('upon', 'works', 'ye')
('works', 'ye', 'mighty')
('ye', 'mighty', 'despair')
```

[ ]:
```
#converts the entire corpus to ngrams
def corpus_to_ngrams(corpus, n, remove_stopwords=True):
    new_corpus = []
    for text in corpus:
        new_corpus.append(text_to_ngrams(text, n, remove_stopwords))
    return new_corpus
```

[ ]:
```
corpus_unigrams = corpus_to_ngrams(corpus, 1)
print(corpus_unigrams[0])
```

```
[('homelessness',), ('houselessness',), ('george',), ('carlin',), ('stated',),
('issue',), ('years',), ('never',), ('plan',), ('help',), ('street',),
('considered',), ('human',), ('everything',), ('going',), ('school',),
('work',), ('vote',), ('matter',), ('people',), ('think',), ('homeless',),
('lost',), ('cause',), ('worrying',), ('things',), ('racism',), ('war',),
('iraq',), ('pressuring',), ('kids',), ('succeed',), ('technology',),
('elections',), ('inflation',), ('worrying',), ("'ll",), ('next',), ('end',),
('streets.',), ('given',), ('bet',), ('live',), ('streets',), ('month',),
('without',), ('luxuries',), ('home',), ('entertainment',), ('sets',),
('bathroom',), ('pictures',), ('wall',), ('computer',), ('everything',),
('treasure',), ('see',), ("'s",), ('like',), ('homeless',), ('goddard',),
('bolt',), ("'s",), ('lesson.',), ('mel',), ('brooks',), ('directs',),
('stars',), ('bolt',), ('plays',), ('rich',), ('man',), ('everything',),
('world',), ('deciding',), ('make',), ('bet',), ('sissy',), ('rival',),
('jeffery',), ('tambor',), ('see',), ('live',), ('streets',), ('thirty',),
('days',), ('without',), ('luxuries',), ('bolt',), ('succeeds',), ('wants',),
('future',), ('project',), ('making',), ('buildings',), ('bet',), ("'s",),
('bolt',), ('thrown',), ('street',), ('bracelet',), ('leg',), ('monitor',),
('every',), ('move',), ('ca',), ("n't",), ('step',), ('sidewalk',), ("'s",),
('given',), ('nickname',), ('pepto',), ('vagrant',), ("'s",), ('written',),
('forehead',), ('bolt',), ('meets',), ('characters',), ('including',),
('woman',), ('name',), ('molly',), ('lesley',), ('ann',), ('warren',), ('ex-
dancer',), ('got',), ('divorce',), ('losing',), ('home',), ('pals',),
('sailor',), ('howard',), ('morris',), ('fumes',), ('teddy',), ('wilson',),
('already',), ('used',), ('streets',), ("'re",), ('survivors',), ('bolt',),
("n't",), ("'s",), ('used',), ('reaching',), ('mutual',), ('agreements',),
('like',), ('rich',), ("'s",), ('fight',), ('flight',), ('kill',), ('killed.',),
('love',), ('connection',), ('molly',), ('bolt',), ("n't",), ('necessary',),
('plot',), ('found',), ('``',), ('life',), ('stinks',), ("''",), ('one',),
('mel',), ('brooks',), ('observant',), ('films',), ('prior',), ('comedy',),
('shows',), ('tender',), ('side',), ('compared',), ('slapstick',), ('work',),
('blazing',), ('saddles',), ('young',), ('frankenstein',), ('spaceballs',),
('matter',), ('show',), ("'s",), ('like',), ('something',), ('valuable',),
('losing',), ('next',), ('day',), ('hand',), ('making',), ('stupid',), ('bet',),
('like',), ('rich',), ('people',), ("n't",), ('know',), ('money',), ('maybe',),
('give',), ('homeless',), ('instead',), ('using',), ('like',), ('monopoly',),
('money.',), ('maybe',), ('film',), ('inspire',), ('help',), ('others',)]
```

[ ]:
```
corpus_bigrams = corpus_to_ngrams(corpus, 2)
print(corpus_bigrams[0])
```

```
[('homelessness', 'houselessness'), ('houselessness', 'george'), ('george',
'carlin'), ('carlin', 'stated'), ('stated', 'issue'), ('issue', 'years'),
('years', 'never'), ('never', 'plan'), ('plan', 'help'), ('help', 'street'),
('street', 'considered'), ('considered', 'human'), ('human', 'everything'),
```

('everything', 'going'), ('going', 'school'), ('school', 'work'), ('work',
'vote'), ('vote', 'matter'), ('matter', 'people'), ('people', 'think'),
('think', 'homeless'), ('homeless', 'lost'), ('lost', 'cause'), ('cause',
'worrying'), ('worrying', 'things'), ('things', 'racism'), ('racism', 'war'),
('war', 'iraq'), ('iraq', 'pressuring'), ('pressuring', 'kids'), ('kids',
'succeed'), ('succeed', 'technology'), ('technology', 'elections'),
('elections', 'inflation'), ('inflation', 'worrying'), ('worrying', "'ll"),
("'ll", 'next'), ('next', 'end'), ('end', 'streets.'), ('streets.', 'given'),
('given', 'bet'), ('bet', 'live'), ('live', 'streets'), ('streets', 'month'),
('month', 'without'), ('without', 'luxuries'), ('luxuries', 'home'), ('home',
'entertainment'), ('entertainment', 'sets'), ('sets', 'bathroom'), ('bathroom',
'pictures'), ('pictures', 'wall'), ('wall', 'computer'), ('computer',
'everything'), ('everything', 'treasure'), ('treasure', 'see'), ('see', "'s"),
("'s", 'like'), ('like', 'homeless'), ('homeless', 'goddard'), ('goddard',
'bolt'), ('bolt', "'s"), ("'s", 'lesson.'), ('lesson.', 'mel'), ('mel',
'brooks'), ('brooks', 'directs'), ('directs', 'stars'), ('stars', 'bolt'),
('bolt', 'plays'), ('plays', 'rich'), ('rich', 'man'), ('man', 'everything'),
('everything', 'world'), ('world', 'deciding'), ('deciding', 'make'), ('make',
'bet'), ('bet', 'sissy'), ('sissy', 'rival'), ('rival', 'jeffery'), ('jeffery',
'tambor'), ('tambor', 'see'), ('see', 'live'), ('live', 'streets'), ('streets',
'thirty'), ('thirty', 'days'), ('days', 'without'), ('without', 'luxuries'),
('luxuries', 'bolt'), ('bolt', 'succeeds'), ('succeeds', 'wants'), ('wants',
'future'), ('future', 'project'), ('project', 'making'), ('making',
'buildings'), ('buildings', 'bet'), ('bet', "'s"), ("'s", 'bolt'), ('bolt',
'thrown'), ('thrown', 'street'), ('street', 'bracelet'), ('bracelet', 'leg'),
('leg', 'monitor'), ('monitor', 'every'), ('every', 'move'), ('move', 'ca'),
('ca', "n't"), ("n't", 'step'), ('step', 'sidewalk'), ('sidewalk', "'s"), ("'s",
'given'), ('given', 'nickname'), ('nickname', 'pepto'), ('pepto', 'vagrant'),
('vagrant', "'s"), ("'s", 'written'), ('written', 'forehead'), ('forehead',
'bolt'), ('bolt', 'meets'), ('meets', 'characters'), ('characters',
'including'), ('including', 'woman'), ('woman', 'name'), ('name', 'molly'),
('molly', 'lesley'), ('lesley', 'ann'), ('ann', 'warren'), ('warren', 'ex-
dancer'), ('ex-dancer', 'got'), ('got', 'divorce'), ('divorce', 'losing'),
('losing', 'home'), ('home', 'pals'), ('pals', 'sailor'), ('sailor', 'howard'),
('howard', 'morris'), ('morris', 'fumes'), ('fumes', 'teddy'), ('teddy',
'wilson'), ('wilson', 'already'), ('already', 'used'), ('used', 'streets'),
('streets', "'re"), ("'re", 'survivors'), ('survivors', 'bolt'), ('bolt',
"n't"), ("n't", "'s"), ("'s", 'used'), ('used', 'reaching'), ('reaching',
'mutual'), ('mutual', 'agreements'), ('agreements', 'like'), ('like', 'rich'),
('rich', "'s"), ("'s", 'fight'), ('fight', 'flight'), ('flight', 'kill'),
('kill', 'killed.'), ('killed.', 'love'), ('love', 'connection'), ('connection',
'molly'), ('molly', 'bolt'), ('bolt', "n't"), ("n't", 'necessary'),
('necessary', 'plot'), ('plot', 'found'), ('found', '``'), ('``', 'life'),
('life', 'stinks'), ('stinks', "''"), ("''", 'one'), ('one', 'mel'), ('mel',
'brooks'), ('brooks', 'observant'), ('observant', 'films'), ('films', 'prior'),
('prior', 'comedy'), ('comedy', 'shows'), ('shows', 'tender'), ('tender',
'side'), ('side', 'compared'), ('compared', 'slapstick'), ('slapstick', 'work'),
('work', 'blazing'), ('blazing', 'saddles'), ('saddles', 'young'), ('young',

```
'frankenstein'), ('frankenstein', 'spaceballs'), ('spaceballs', 'matter'),
('matter', 'show'), ('show', "'s"), ("'s", 'like'), ('like', 'something'),
('something', 'valuable'), ('valuable', 'losing'), ('losing', 'next'), ('next',
'day'), ('day', 'hand'), ('hand', 'making'), ('making', 'stupid'), ('stupid',
'bet'), ('bet', 'like'), ('like', 'rich'), ('rich', 'people'), ('people',
"n't"), ("n't", 'know'), ('know', 'money'), ('money', 'maybe'), ('maybe',
'give'), ('give', 'homeless'), ('homeless', 'instead'), ('instead', 'using'),
('using', 'like'), ('like', 'monopoly'), ('monopoly', 'money.'), ('money.',
'maybe'), ('maybe', 'film'), ('film', 'inspire'), ('inspire', 'help'), ('help',
'others')]
```

[ ]:
```
corpus_trigrams = corpus_to_ngrams(corpus, 3)
print(corpus_trigrams[0])
```

```
[('homelessness', 'houselessness', 'george'), ('houselessness', 'george',
'carlin'), ('george', 'carlin', 'stated'), ('carlin', 'stated', 'issue'),
('stated', 'issue', 'years'), ('issue', 'years', 'never'), ('years', 'never',
'plan'), ('never', 'plan', 'help'), ('plan', 'help', 'street'), ('help',
'street', 'considered'), ('street', 'considered', 'human'), ('considered',
'human', 'everything'), ('human', 'everything', 'going'), ('everything',
'going', 'school'), ('going', 'school', 'work'), ('school', 'work', 'vote'),
('work', 'vote', 'matter'), ('vote', 'matter', 'people'), ('matter', 'people',
'think'), ('people', 'think', 'homeless'), ('think', 'homeless', 'lost'),
('homeless', 'lost', 'cause'), ('lost', 'cause', 'worrying'), ('cause',
'worrying', 'things'), ('worrying', 'things', 'racism'), ('things', 'racism',
'war'), ('racism', 'war', 'iraq'), ('war', 'iraq', 'pressuring'), ('iraq',
'pressuring', 'kids'), ('pressuring', 'kids', 'succeed'), ('kids', 'succeed',
'technology'), ('succeed', 'technology', 'elections'), ('technology',
'elections', 'inflation'), ('elections', 'inflation', 'worrying'), ('inflation',
'worrying', "'ll"), ('worrying', "'ll", 'next'), ("'ll", 'next', 'end'),
('next', 'end', 'streets.'), ('end', 'streets.', 'given'), ('streets.', 'given',
'bet'), ('given', 'bet', 'live'), ('bet', 'live', 'streets'), ('live',
'streets', 'month'), ('streets', 'month', 'without'), ('month', 'without',
'luxuries'), ('without', 'luxuries', 'home'), ('luxuries', 'home',
'entertainment'), ('home', 'entertainment', 'sets'), ('entertainment', 'sets',
'bathroom'), ('sets', 'bathroom', 'pictures'), ('bathroom', 'pictures', 'wall'),
('pictures', 'wall', 'computer'), ('wall', 'computer', 'everything'),
('computer', 'everything', 'treasure'), ('everything', 'treasure', 'see'),
('treasure', 'see', "'s"), ('see', "'s", 'like'), ("'s", 'like', 'homeless'),
('like', 'homeless', 'goddard'), ('homeless', 'goddard', 'bolt'), ('goddard',
'bolt', "'s"), ('bolt', "'s", 'lesson.'), ("'s", 'lesson.', 'mel'), ('lesson.',
'mel', 'brooks'), ('mel', 'brooks', 'directs'), ('brooks', 'directs', 'stars'),
('directs', 'stars', 'bolt'), ('stars', 'bolt', 'plays'), ('bolt', 'plays',
'rich'), ('plays', 'rich', 'man'), ('rich', 'man', 'everything'), ('man',
'everything', 'world'), ('everything', 'world', 'deciding'), ('world',
'deciding', 'make'), ('deciding', 'make', 'bet'), ('make', 'bet', 'sissy'),
('bet', 'sissy', 'rival'), ('sissy', 'rival', 'jeffery'), ('rival', 'jeffery',
'tambor'), ('jeffery', 'tambor', 'see'), ('tambor', 'see', 'live'), ('see',
```

'live', 'streets'), ('live', 'streets', 'thirty'), ('streets', 'thirty',
'days'), ('thirty', 'days', 'without'), ('days', 'without', 'luxuries'),
('without', 'luxuries', 'bolt'), ('luxuries', 'bolt', 'succeeds'), ('bolt',
'succeeds', 'wants'), ('succeeds', 'wants', 'future'), ('wants', 'future',
'project'), ('future', 'project', 'making'), ('project', 'making', 'buildings'),
('making', 'buildings', 'bet'), ('buildings', 'bet', "'s"), ('bet', "'s",
'bolt'), ("'s", 'bolt', 'thrown'), ('bolt', 'thrown', 'street'), ('thrown',
'street', 'bracelet'), ('street', 'bracelet', 'leg'), ('bracelet', 'leg',
'monitor'), ('leg', 'monitor', 'every'), ('monitor', 'every', 'move'), ('every',
'move', 'ca'), ('move', 'ca', "n't"), ('ca', "n't", 'step'), ("n't", 'step',
'sidewalk'), ('step', 'sidewalk', "'s"), ('sidewalk', "'s", 'given'), ("'s",
'given', 'nickname'), ('given', 'nickname', 'pepto'), ('nickname', 'pepto',
'vagrant'), ('pepto', 'vagrant', "'s"), ('vagrant', "'s", 'written'), ("'s",
'written', 'forehead'), ('written', 'forehead', 'bolt'), ('forehead', 'bolt',
'meets'), ('bolt', 'meets', 'characters'), ('meets', 'characters', 'including'),
('characters', 'including', 'woman'), ('including', 'woman', 'name'), ('woman',
'name', 'molly'), ('name', 'molly', 'lesley'), ('molly', 'lesley', 'ann'),
('lesley', 'ann', 'warren'), ('ann', 'warren', 'ex-dancer'), ('warren', 'ex-
dancer', 'got'), ('ex-dancer', 'got', 'divorce'), ('got', 'divorce', 'losing'),
('divorce', 'losing', 'home'), ('losing', 'home', 'pals'), ('home', 'pals',
'sailor'), ('pals', 'sailor', 'howard'), ('sailor', 'howard', 'morris'),
('howard', 'morris', 'fumes'), ('morris', 'fumes', 'teddy'), ('fumes', 'teddy',
'wilson'), ('teddy', 'wilson', 'already'), ('wilson', 'already', 'used'),
('already', 'used', 'streets'), ('used', 'streets', "'re"), ('streets', "'re",
'survivors'), ("'re", 'survivors', 'bolt'), ('survivors', 'bolt', "n't"),
('bolt', "n't", "'s"), ("n't", "'s", 'used'), ("'s", 'used', 'reaching'),
('used', 'reaching', 'mutual'), ('reaching', 'mutual', 'agreements'), ('mutual',
'agreements', 'like'), ('agreements', 'like', 'rich'), ('like', 'rich', "'s"),
('rich', "'s", 'fight'), ("'s", 'fight', 'flight'), ('fight', 'flight', 'kill'),
('flight', 'kill', 'killed.'), ('kill', 'killed.', 'love'), ('killed.', 'love',
'connection'), ('love', 'connection', 'molly'), ('connection', 'molly', 'bolt'),
('molly', 'bolt', "n't"), ('bolt', "n't", 'necessary'), ("n't", 'necessary',
'plot'), ('necessary', 'plot', 'found'), ('plot', 'found', '``'), ('found',
'``', 'life'), ('``', 'life', 'stinks'), ('life', 'stinks', "'`"), ('stinks',
"'`", 'one'), ("'`", 'one', 'mel'), ('one', 'mel', 'brooks'), ('mel', 'brooks',
'observant'), ('brooks', 'observant', 'films'), ('observant', 'films', 'prior'),
('films', 'prior', 'comedy'), ('prior', 'comedy', 'shows'), ('comedy', 'shows',
'tender'), ('shows', 'tender', 'side'), ('tender', 'side', 'compared'), ('side',
'compared', 'slapstick'), ('compared', 'slapstick', 'work'), ('slapstick',
'work', 'blazing'), ('work', 'blazing', 'saddles'), ('blazing', 'saddles',
'young'), ('saddles', 'young', 'frankenstein'), ('young', 'frankenstein',
'spaceballs'), ('frankenstein', 'spaceballs', 'matter'), ('spaceballs',
'matter', 'show'), ('matter', 'show', "'s"), ('show', "'s", 'like'), ("'s",
'like', 'something'), ('like', 'something', 'valuable'), ('something',
'valuable', 'losing'), ('valuable', 'losing', 'next'), ('losing', 'next',
'day'), ('next', 'day', 'hand'), ('day', 'hand', 'making'), ('hand', 'making',
'stupid'), ('making', 'stupid', 'bet'), ('stupid', 'bet', 'like'), ('bet',
'like', 'rich'), ('like', 'rich', 'people'), ('rich', 'people', "n't"),

```
('people', "n't", 'know'), ("n't", 'know', 'money'), ('know', 'money', 'maybe'),
('money', 'maybe', 'give'), ('maybe', 'give', 'homeless'), ('give', 'homeless',
'instead'), ('homeless', 'instead', 'using'), ('instead', 'using', 'like'),
('using', 'like', 'monopoly'), ('like', 'monopoly', 'money.'), ('monopoly',
'money.', 'maybe'), ('money.', 'maybe', 'film'), ('maybe', 'film', 'inspire'),
('film', 'inspire', 'help'), ('inspire', 'help', 'others')]
```

[ ]:
```
corps_unigrams_with_stopwords = corpus_to_ngrams(corpus, 1,␣
↪remove_stopwords=False)
print(corps_unigrams_with_stopwords[0])
```

```
[('homelessness',), ('or',), ('houselessness',), ('as',), ('george',),
('carlin',), ('stated',), ('has',), ('been',), ('an',), ('issue',), ('for',),
('years',), ('but',), ('never',), ('a',), ('plan',), ('to',), ('help',),
('those',), ('on',), ('the',), ('street',), ('that',), ('were',), ('once',),
('considered',), ('human',), ('who',), ('did',), ('everything',), ('from',),
('going',), ('to',), ('school',), ('work',), ('or',), ('vote',), ('for',),
('the',), ('matter',), ('most',), ('people',), ('think',), ('of',), ('the',),
('homeless',), ('as',), ('just',), ('a',), ('lost',), ('cause',), ('while',),
('worrying',), ('about',), ('things',), ('such',), ('as',), ('racism',),
('the',), ('war',), ('on',), ('iraq',), ('pressuring',), ('kids',), ('to',),
('succeed',), ('technology',), ('the',), ('elections',), ('inflation',),
('or',), ('worrying',), ('if',), ('they',), ("'ll",), ('be',), ('next',),
('to',), ('end',), ('up',), ('on',), ('the',), ('streets.',), ('but',),
('what',), ('if',), ('you',), ('were',), ('given',), ('a',), ('bet',), ('to',),
('live',), ('on',), ('the',), ('streets',), ('for',), ('a',), ('month',),
('without',), ('the',), ('luxuries',), ('you',), ('once',), ('had',), ('from',),
('a',), ('home',), ('the',), ('entertainment',), ('sets',), ('a',),
('bathroom',), ('pictures',), ('on',), ('the',), ('wall',), ('a',),
('computer',), ('and',), ('everything',), ('you',), ('once',), ('treasure',),
('to',), ('see',), ('what',), ('it',), ("'s",), ('like',), ('to',), ('be',),
('homeless',), ('that',), ('is',), ('goddard',), ('bolt',), ("'s",),
('lesson.',), ('mel',), ('brooks',), ('who',), ('directs',), ('who',),
('stars',), ('as',), ('bolt',), ('plays',), ('a',), ('rich',), ('man',),
('who',), ('has',), ('everything',), ('in',), ('the',), ('world',), ('until',),
('deciding',), ('to',), ('make',), ('a',), ('bet',), ('with',), ('a',),
('sissy',), ('rival',), ('jeffery',), ('tambor',), ('to',), ('see',), ('if',),
('he',), ('can',), ('live',), ('in',), ('the',), ('streets',), ('for',),
('thirty',), ('days',), ('without',), ('the',), ('luxuries',), ('if',),
('bolt',), ('succeeds',), ('he',), ('can',), ('do',), ('what',), ('he',),
('wants',), ('with',), ('a',), ('future',), ('project',), ('of',), ('making',),
('more',), ('buildings',), ('the',), ('bet',), ("'s",), ('on',), ('where',),
('bolt',), ('is',), ('thrown',), ('on',), ('the',), ('street',), ('with',),
('a',), ('bracelet',), ('on',), ('his',), ('leg',), ('to',), ('monitor',),
('his',), ('every',), ('move',), ('where',), ('he',), ('ca',), ("n't",),
('step',), ('off',), ('the',), ('sidewalk',), ('he',), ("'s",), ('given',),
('the',), ('nickname',), ('pepto',), ('by',), ('a',), ('vagrant',), ('after',),
('it',), ("'s",), ('written',), ('on',), ('his',), ('forehead',), ('where',),
```

```
('bolt',), ('meets',), ('other',), ('characters',), ('including',), ('a',),
('woman',), ('by',), ('the',), ('name',), ('of',), ('molly',), ('lesley',),
('ann',), ('warren',), ('an',), ('ex-dancer',), ('who',), ('got',),
('divorce',), ('before',), ('losing',), ('her',), ('home',), ('and',), ('her',),
('pals',), ('sailor',), ('howard',), ('morris',), ('and',), ('fumes',),
('teddy',), ('wilson',), ('who',), ('are',), ('already',), ('used',), ('to',),
('the',), ('streets',), ('they',), ("'re",), ('survivors',), ('bolt',), ('is',),
("n't",), ('he',), ("'s",), ('not',), ('used',), ('to',), ('reaching',),
('mutual',), ('agreements',), ('like',), ('he',), ('once',), ('did',),
('when',), ('being',), ('rich',), ('where',), ('it',), ("'s",), ('fight',),
('or',), ('flight',), ('kill',), ('or',), ('be',), ('killed.',), ('while',),
('the',), ('love',), ('connection',), ('between',), ('molly',), ('and',),
('bolt',), ('was',), ("n't",), ('necessary',), ('to',), ('plot',), ('i',),
('found',), ('``',), ('life',), ('stinks',), ("''",), ('to',), ('be',),
('one',), ('of',), ('mel',), ('brooks',), ('observant',), ('films',),
('where',), ('prior',), ('to',), ('being',), ('a',), ('comedy',), ('it',),
('shows',), ('a',), ('tender',), ('side',), ('compared',), ('to',), ('his',),
('slapstick',), ('work',), ('such',), ('as',), ('blazing',), ('saddles',),
('young',), ('frankenstein',), ('or',), ('spaceballs',), ('for',), ('the',),
('matter',), ('to',), ('show',), ('what',), ('it',), ("'s",), ('like',),
('having',), ('something',), ('valuable',), ('before',), ('losing',), ('it',),
('the',), ('next',), ('day',), ('or',), ('on',), ('the',), ('other',),
('hand',), ('making',), ('a',), ('stupid',), ('bet',), ('like',), ('all',),
('rich',), ('people',), ('do',), ('when',), ('they',), ('do',), ("n't",),
('know',), ('what',), ('to',), ('do',), ('with',), ('their',), ('money',),
('maybe',), ('they',), ('should',), ('give',), ('it',), ('to',), ('the',),
('homeless',), ('instead',), ('of',), ('using',), ('it',), ('like',),
('monopoly',), ('money.',), ('or',), ('maybe',), ('this',), ('film',),
('will',), ('inspire',), ('you',), ('to',), ('help',), ('others',)]
```

Feature Selection using Lemmatization and Stemming

[ ]:
```
def apply_stemming(text):
    st = LancasterStemmer()
    word_list = [" ".join(st.stem(gram) for gram in ngram) for ngram in text]
                # stems the list of ngram tuples using nltk's LancasterStemmer
    return word_list
```

[ ]:
```
stemmed_text = apply_stemming(grams)
for feature in stemmed_text:
    print(feature)
```

```
ozymandia king king
king king look
king look upon
look upon work
upon work ye
work ye mighty
ye mighty despair
```

[ ]:
```
def apply_lemmatization(text):
    lm = WordNetLemmatizer()
    word_list = [" ".join(lm.lemmatize(gram) for gram in ngram) for ngram in␣
 ↪text]
                # lemmatizes the list of ngram tuples
    return word_list
```

[ ]:
```
lemmatized_text = apply_lemmatization(grams)
for feature in lemmatized_text:
    print(feature)
```

```
ozymandias king king
king king look
king look upon
look upon work
upon work ye
work ye mighty
ye mighty despair
```

[ ]:
```
#apply a given stemming or lemmatization function to the corpus
def apply_to_corpus(func, corpus):
    new_corpus = []
    for text in corpus:
        new_corpus.append(func(text))
    return new_corpus
```

[ ]:
```
lemmatized_unigrams = apply_to_corpus(apply_lemmatization, corpus_unigrams)
stemmed_unigrams = apply_to_corpus(apply_stemming, corpus_unigrams)
print(lemmatized_unigrams[0])
print(stemmed_unigrams[0])
```

```
['homelessness', 'houselessness', 'george', 'carlin', 'stated', 'issue', 'year',
'never', 'plan', 'help', 'street', 'considered', 'human', 'everything', 'going',
'school', 'work', 'vote', 'matter', 'people', 'think', 'homeless', 'lost',
'cause', 'worrying', 'thing', 'racism', 'war', 'iraq', 'pressuring', 'kid',
'succeed', 'technology', 'election', 'inflation', 'worrying', "'ll", 'next',
'end', 'streets.', 'given', 'bet', 'live', 'street', 'month', 'without',
'luxury', 'home', 'entertainment', 'set', 'bathroom', 'picture', 'wall',
'computer', 'everything', 'treasure', 'see', "'s", 'like', 'homeless',
'goddard', 'bolt', "'s", 'lesson.', 'mel', 'brook', 'directs', 'star', 'bolt',
'play', 'rich', 'man', 'everything', 'world', 'deciding', 'make', 'bet',
'sissy', 'rival', 'jeffery', 'tambor', 'see', 'live', 'street', 'thirty', 'day',
'without', 'luxury', 'bolt', 'succeeds', 'want', 'future', 'project', 'making',
'building', 'bet', "'s", 'bolt', 'thrown', 'street', 'bracelet', 'leg',
'monitor', 'every', 'move', 'ca', "n't", 'step', 'sidewalk', "'s", 'given',
'nickname', 'pepto', 'vagrant', "'s", 'written', 'forehead', 'bolt', 'meet',
'character', 'including', 'woman', 'name', 'molly', 'lesley', 'ann', 'warren',
'ex-dancer', 'got', 'divorce', 'losing', 'home', 'pal', 'sailor', 'howard',
```

```
'morris', 'fume', 'teddy', 'wilson', 'already', 'used', 'street', "'re",
'survivor', 'bolt', "n't", "'s", 'used', 'reaching', 'mutual', 'agreement',
'like', 'rich', "'s", 'fight', 'flight', 'kill', 'killed.', 'love',
'connection', 'molly', 'bolt', "n't", 'necessary', 'plot', 'found', '``',
'life', 'stink', "'", 'one', 'mel', 'brook', 'observant', 'film', 'prior',
'comedy', 'show', 'tender', 'side', 'compared', 'slapstick', 'work', 'blazing',
'saddle', 'young', 'frankenstein', 'spaceballs', 'matter', 'show', "'s", 'like',
'something', 'valuable', 'losing', 'next', 'day', 'hand', 'making', 'stupid',
'bet', 'like', 'rich', 'people', "n't", 'know', 'money', 'maybe', 'give',
'homeless', 'instead', 'using', 'like', 'monopoly', 'money.', 'maybe', 'film',
'inspire', 'help', 'others']
['homeless', 'houseless', 'georg', 'carlin', 'stat', 'issu', 'year', 'nev',
'plan', 'help', 'street', 'consid', 'hum', 'everyth', 'going', 'school', 'work',
'vot', 'mat', 'peopl', 'think', 'homeless', 'lost', 'caus', 'worry', 'thing',
'rac', 'war', 'iraq', 'press', 'kid', 'success', 'technolog', 'elect', 'infl',
'worry', "'ll", 'next', 'end', 'streets.', 'giv', 'bet', 'liv', 'streets',
'mon', 'without', 'luxury', 'hom', 'entertain', 'set', 'bathroom', 'pict',
'wal', 'comput', 'everyth', 'treas', 'see', "'s", 'lik', 'homeless', 'goddard',
'bolt', "'s", 'lesson.', 'mel', 'brook', 'direct', 'star', 'bolt', 'play',
'rich', 'man', 'everyth', 'world', 'decid', 'mak', 'bet', 'sissy', 'riv',
'jeffery', 'tamb', 'see', 'liv', 'streets', 'thirty', 'day', 'without',
'luxury', 'bolt', 'success', 'want', 'fut', 'project', 'mak', 'build', 'bet',
"'s", 'bolt', 'thrown', 'street', 'bracelet', 'leg', 'monit', 'every', 'mov',
'ca', "n't", 'step', 'sidewalk', "'s", 'giv', 'nicknam', 'pepto', 'vagr', "'s",
'writ', 'forehead', 'bolt', 'meet', 'charact', 'includ', 'wom', 'nam', 'mol',
'lesley', 'an', 'war', 'ex-dancer', 'got', 'divorc', 'los', 'hom', 'pal',
'sail', 'howard', 'mor', 'fum', 'teddy', 'wilson', 'already', 'us', 'streets',
"'re", 'surv', 'bolt', "n't", "'s", 'us', 'reach', 'mut', 'agr', 'lik', 'rich',
"'s", 'fight', 'flight', 'kil', 'killed.', 'lov', 'connect', 'mol', 'bolt',
"n't", 'necess', 'plot', 'found', '``', 'lif', 'stink', "'", 'on', 'mel',
'brook', 'observ', 'film', 'pri', 'comedy', 'show', 'tend', 'sid', 'comp',
'slapstick', 'work', 'blaz', 'saddl', 'young', 'frankenstein', 'spacebal',
'mat', 'show', "'s", 'lik', 'someth', 'valu', 'los', 'next', 'day', 'hand',
'mak', 'stupid', 'bet', 'lik', 'rich', 'peopl', "n't", 'know', 'money', 'mayb',
'giv', 'homeless', 'instead', 'us', 'lik', 'monopo', 'money.', 'mayb', 'film',
'inspir', 'help', 'oth']
```

TF-IDF

First we set about generating a shared vocabulary, containing the number of documents each unique word occurs in, in order to calculate TF and IDF values.

[ ]:
```
def generate_shared_vocabulary(corpus):
    words = {}
    for text in corpus:
        for word in set(text):
            # set(text) removes duplicates, meaning the dictionary contains␣
 ↪document frequency values
```

```
            # (number of documents our word occurs in)
            if word in words:
                words[word] += 1
            else:
                words[word] = 1
    return words
```

[ ]:
```
shared_vocabulary = generate_shared_vocabulary(stemmed_unigrams)
```

Now we generate our TF-IDF matrix, where each row represents a document in the corpus. We utilise the scipy sparse matrix data structure in order to save memory.

[ ]:
```
from scipy import sparse

def generate_tf_idf_matrix(corpus, shared_vocabulary, one_hot=False):
    N = len(shared_vocabulary)
    shared_vocabulary_list = list(shared_vocabulary)
    matrix = sparse.lil_matrix(np.zeros([corpus_length, N]))
    #sparse list of lists to store our tf_idf values

    for i, text in enumerate(corpus):
        for word in text: # calculate tf_idf for each feature in each document
                          # and insert in correct index
            index = shared_vocabulary_list.index(word)
            tf = text.count(word) / len(text)
            idf = np.log10(N / (shared_vocabulary[word] +1))
            matrix[i, index] = tf * idf if not one_hot else 1
            #if using one_hot vectors for SVM and LogReg then simply insert 1
    return matrix
```

[ ]:
```
stem_unigram_tf_idf = generate_tf_idf_matrix(stemmed_unigrams,␣
↪shared_vocabulary)
stem_unigram_tf_idf[0].toarray()
```

[ ]:
```
array([[0.01005802, 0.00693199, 0.01359507, …, 0.        , 0.        ,
        0.        ]])
```

Splitting the data

[ ]:
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.model_selection import train_test_split

r_seed = 563
np.random.seed(r_seed)

def get_test_train_dev_split(X):
    y = np.concatenate([np.ones(positive_labels), np.zeros(negative_labels)])
```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.80,
                                                        shuffle=True,␣
 ↪random_state=r_seed)

    X_train, X_dev, y_train, y_dev = train_test_split(X_train, y_train,
                                                      test_size=0.15,
                                                      shuffle=True,␣
 ↪random_state=r_seed)

    #we first split the data into train and test, then we split train into␣
 ↪train
    #and development
    #68% train, 12% validation, 20% test
    return X_train, y_train, X_test, y_test, X_dev, y_dev

X_train, y_train, X_test, y_test, X_dev, y_dev =␣
 ↪get_test_train_dev_split(stem_unigram_tf_idf)
print(X_train.shape)
print(X_test.shape)
print(X_dev.shape)
print(X_train.toarray()[0])
```

```
(2720, 29379)
(800, 29379)
(480, 29379)
[0.        0.0105175 0.        … 0.        0.        0.       ]
```

Multinomial Naive Bayes

[ ]:
```
'''
calculate_likelihood: calculates the likelihood that a feature x belongs to␣
 ↪class C, p(x|C)
labels: 1 for the class whose likelihood is being calculated, 0 for any others
data: the training data, i.e. our TF-IDF matrix of all documents
alpha: the alphas value for laplace smoothing
'''
def calculate_likelihoods(data, labels, alpha=1.0):
    N = data.shape[1]
    likelihoods = np.zeros([N])
    for i in range(N):
        feature = data[:, i].toarray().flatten()
        likelihoods[i] = (np.sum(feature * labels) + alpha)  / (np.sum(labels)␣
 ↪+ alpha)
        # likelihood calculation (p(X|C)) using laplace smoothing
    return likelihoods
```

[ ]:
```
'''
#calculates the likelihoods for both classes given the training data
#as well as priors for both classes
#inverted_y_train: an inverted label array, denoting 1 for the negative class␣
 ↪and 0 for the positive
#used in calculating likelihood and priors
'''
def train_multinomial_bayes(X_train, y_train):
    inverted_y_train = np.array([not y for y in y_train]).astype(int)

    pos_likelihoods = calculate_likelihoods(X_train, y_train)
    neg_likelihoods = calculate_likelihoods(X_train, inverted_y_train)
    pos_log_likelihoods = np.log(pos_likelihoods)
    neg_log_likelihoods = np.log(neg_likelihoods)

    pos_prior = np.sum(y_train) / len(y_train)
    neg_prior = np.sum(inverted_y_train) / len(inverted_y_train)
    pos_log_prior = np.log(pos_prior)
    neg_log_prior = np.log(neg_prior)
    return pos_log_likelihoods, neg_log_likelihoods, pos_log_prior,␣
 ↪neg_log_prior
```

[ ]:
```
#assigns a class label to a given document using likelihoods and priors
def get_multinomial_class_label(data, document):
    pos_log_likelihoods, neg_log_likelihoods, pos_log_prior, neg_log_prior =␣
 ↪data
    #unpacks our sparse vector:
    features = np.nonzero(document)[0]
    pos_total = 0
    neg_total = 0
    for index in features:
        #sum log likelihoods for each feature, for both classes
        pos_total += pos_log_likelihoods[index]
        neg_total += neg_log_likelihoods[index]
    #add priors
    pos_total += pos_log_prior
    neg_total += neg_log_prior
    class_label = 1 if pos_total > neg_total else 0
    return class_label
```

[ ]:
```
#runs the entire pipeline for MNB and returns a predictions array
def test_train_multinomial_bayes(train_data, train_labels, test_data):
    data = train_multinomial_bayes(train_data, train_labels)
    predictions = []
    for i, v in enumerate(test_data):
        doc =  test_data[i].toarray().flatten()
```

```
        label = get_multinomial_class_label(data, doc)
        predictions.append(label)
    return predictions
```

[ ]:
```
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score

def evaluate_model( test_labels, predictions):
    print("accuracy:", accuracy_score(test_labels, predictions))
    print("precision:", precision_score(test_labels, predictions))
    print("recall:", recall_score(test_labels, predictions))
    print("f1 score:", f1_score(test_labels, predictions))
    print()
```

Evaluation for Multinomial Naive Bayes on the development set for stemmed unigrams:

[ ]:
```
predictions = test_train_multinomial_bayes(X_train, y_train, X_dev)
evaluate_model(y_dev, predictions)
```

```
0.5025735294117647 0.4974264705882353
accuracy: 0.875
precision: 0.9267015706806283
recall: 0.7937219730941704
f1 score: 0.8550724637681159
```

Gaussian Naive Bayes

[ ]:
```
'''
calculate_guassian_distributions: calculates the mean and standard distribution␣
 ↪for
a given feature using TF-IDF scores across all documents
labels: 1 for the class whose likelihood is being calculated, 0 for any others
data: the training data, i.e. our TF-IDF matrix of all documents
alpha: the alphas value for laplace smoothing
'''
def calculate_guassian_distributions(data, labels, alpha=1e-10):
    pos_distribution = []
    neg_distribution = []
    inverted_labels = np.array([not y for y in labels]).astype(int)

    for i in range(data.shape[1]): #calculate means and standard deviations for␣
 ↪each feature
                                   #in order to compute distributions
        feature = data[:, i].toarray().flatten()
```

```
        #collects the instances of the feature being present in a positive and␣
 ↪negative class resp.
        pos_feature = feature * labels
        neg_feature = feature * inverted_labels
        pos_distribution.append((np.mean(pos_feature) + alpha, np.
 ↪std(pos_feature) + alpha))
        neg_distribution.append((np.mean(neg_feature) + alpha, np.
 ↪std(neg_feature) + alpha))
    return pos_distribution, neg_distribution
```

[ ]:
```
'''
#calculates the likelihoods for both classes given the training data
#as well as priors for both classes
#inverted_y_train: an inverted label array, denoting 1 for the negative class␣
 ↪and 0 for the positive
#used in calculating likelihood and priors
'''
def train_gaussian_bayes(X_train, y_train):
    inverted_y_train = np.array([not y for y in y_train]).astype(int)

    pos_distribution, neg_distribution =␣
 ↪calculate_guassian_distributions(X_train, y_train)
    pos_log_distribution = np.log(pos_distribution)
    neg_log_distribution = np.log(neg_distribution)

    pos_prior = np.sum(y_train) / len(y_train)
    neg_prior = np.sum(inverted_y_train) / len(inverted_y_train)
    pos_log_prior = np.log(pos_prior)
    neg_log_prior = np.log(neg_prior)

    return pos_log_distribution, neg_log_distribution, pos_log_prior,␣
 ↪neg_log_prior
```

[ ]:
```
#fits value (the TF-IDF score for a given feature in the input document) to the␣
 ↪guassian distribution of said feature
def gaussian(mean, sd, value):
    exponent = (- (value - mean)**2 ) / (2 * sd**2)
    value = (1 / np.sqrt(2 * np.pi * sd**2)) * np.exp(exponent)
    if np.isnan(value): #0 if NaN
        return 0
    return value
```

[ ]:
```
#produces a class label for a given document using our GNB
def get_gaussian_class_label(data, document):
    pos_distribution, neg_distribution, pos_log_prior, neg_log_prior = data
    features = np.nonzero(document)[0]
    pos_total = 0
```

```
    neg_total = 0
    for index in features: #calculates the likelihood using the mean, sd, and␣
↪value for each feature
                           #gaussian(mean, sd, x)
                           #pos_distribution[i] = (mean, sd), where i is␣
↪feature index
        pos_total += gaussian(pos_distribution[index][0],␣
↪pos_distribution[index][1],
                              document[index] )
        neg_total += gaussian(neg_distribution[index][0],␣
↪neg_distribution[index][1],
                              document[index] )
    pos_total += pos_log_prior
    neg_total += neg_log_prior
    label = 1 if pos_total > neg_total else 0
    return label
```

[ ]:
```
#full GNB pipeline
def test_train_gaussian_bayes(train_data, train_labels, test_data):
    data = train_gaussian_bayes(train_data, train_labels)
    predictions = []
    for i, v in enumerate(test_data):
        doc =  test_data[i].toarray().flatten()
        label = get_gaussian_class_label(data, doc)
        predictions.append(label)
    return predictions
```

Evaluation on development set for Guassian Bayes using stemmed unigrams

[ ]:
```
predictions = test_train_gaussian_bayes(X_train, y_train, X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8645833333333334
precision: 0.8291666666666667
recall: 0.8923766816143498
f1 score: 0.8596112311015119
```

Sklearn MNB and GNB Models

[ ]:
```
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import GaussianNB

def test_train_sklearn_models(X_train, y_train, X_test, y_test):
    clf = MultinomialNB()
    clf.fit(X_train, y_train)
    predictions = clf.predict(X_test)
```

```
    print("Sklearn Multinomial Bayes")
    evaluate_model(y_test, predictions)

    clf2 = GaussianNB()
    clf2.fit(X_train.toarray(), y_train)
    predictions = clf2.predict(X_test.toarray())
    print("Sklearn Gaussian Bayes")
    evaluate_model(y_test, predictions)
test_train_sklearn_models(X_train, y_train, X_dev, y_dev)
```

```
Sklearn Multinomial Bayes
accuracy: 0.85625
precision: 0.8155737704918032
recall: 0.8923766816143498
f1 score: 0.8522483940042827

Sklearn Gaussian Bayes
accuracy: 0.65
precision: 0.611336032388664
recall: 0.6771300448430493
f1 score: 0.6425531914893617
```

For stemmed unigrams our own implementation of multinomial bayes achieves a similar accuracy and f1score, higher precision and a lower recall in comparison to the prebuilt sklearn model. Own implementation of Gaussian Bayes far outperforms on accuracy, precision, and f1 score.

Evaluation

[ ]:
```
#full training and evaluation for a given corpus using all models
def evaluate_on_corpus(corpus):
    shared_vocabulary = generate_shared_vocabulary(corpus)
    tf_idf_matrix = generate_tf_idf_matrix(corpus, shared_vocabulary)
    X_train, y_train, X_test, y_test, X_dev, y_dev =␣
 ↪get_test_train_dev_split(tf_idf_matrix)

    multinomial_predictions = test_train_multinomial_bayes(X_train, y_train,␣
 ↪X_dev)
    print("Multinomial Bayes")
    evaluate_model(y_dev, multinomial_predictions)
    gaussian_predictions = test_train_gaussian_bayes(X_train, y_train, X_dev)
    print("Gaussian Bayes")
    evaluate_model(y_dev, gaussian_predictions)
    test_train_sklearn_models(X_train, y_train, X_dev, y_dev)
    return tf_idf_matrix
```

Note that we ran stemmed and lemmatized bigrams and produced the following results. These are omitted as code cells due to the time taken to run (> 90 minutes).

lemmatized_bigrams = apply_to_corpus(apply_lemmatization, corpus_bigrams) evalu-

ate_on_corpus(lemmatized_bigrams) Lemmatized Bigrams: Multinomial Bayes accuracy: 0.555 precision: 0.9666666666666667 recall: 0.07552083333333333 f1 score: 0.1400966183574879 Gaussian Bayes accuracy: 0.74375 precision: 0.6660482374768089 recall: 0.9348958333333334 f1 score: 0.7778981581798483 Stemmed Bigrams: Multinomial Bayes accuracy: 0.6375 precision: 0.8955223880597015 recall: 0.2643171806167401 f1 score: 0.40816326530612246 Gaussian Bayes accuracy: 0.7729166666666667 precision: 0.7341269841269841 recall: 0.8149779735682819 f1 score: 0.7724425887265136 Sklearn Multinomial Bayes accuracy: 0.7958333333333333 precision: 0.7698744769874477 recall: 0.8105726872246696 f1 score: 0.7896995708154506 Sklearn Gaussian Bayes accuracy: 0.7229166666666667 precision: 0.706140350877193 recall: 0.7092511013215859 f1 score: 0.7076923076923076

Having tried stemmed unigrams let's asses the use of lemmatized unigrams:

[ ]:
```
lemmatized_unigrams = apply_to_corpus(apply_lemmatization, corpus_unigrams)
lem_unigrams_tf_idf = evaluate_on_corpus(lemmatized_unigrams)
```

```
Multinomial Bayes
accuracy: 0.8666666666666667
precision: 0.9392265193370166
recall: 0.7623318385650224
f1 score: 0.8415841584158416

Gaussian Bayes
accuracy: 0.8666666666666667
precision: 0.8326359832635983
recall: 0.8923766816143498
f1 score: 0.8614718614718615

Sklearn Multinomial Bayes
accuracy: 0.8645833333333334
precision: 0.8464912280701754
recall: 0.8654708520179372
f1 score: 0.8558758314855874

Sklearn Gaussian Bayes
accuracy: 0.6354166666666666
precision: 0.5930232558139535
recall: 0.6860986547085202
f1 score: 0.6361746361746361
```

Clearly stemming achieves higher performance across the board. Let's try stemming without stopword removal and assess the performance

[ ]:
```
stemmed_unigrams_stopwords = apply_to_corpus(apply_stemming,␣
↪corps_unigrams_with_stopwords)
stem_uni_stopword_tf_idf = evaluate_on_corpus(stemmed_unigrams_stopwords)
```

```
Multinomial Bayes
accuracy: 0.8020833333333334
```

```
precision: 0.9266666666666666
recall: 0.6233183856502242
f1 score: 0.7453083109919572

Gaussian Bayes
accuracy: 0.8604166666666667
precision: 0.8305084745762712
recall: 0.8789237668161435
f1 score: 0.8540305010893247

Sklearn Multinomial Bayes
accuracy: 0.8354166666666667
precision: 0.7790697674418605
recall: 0.9013452914798207
f1 score: 0.8357588357588358

Sklearn Gaussian Bayes
accuracy: 0.65
precision: 0.6122448979591837
recall: 0.672645739910314
f1 score: 0.6410256410256411
```

Thus our best feature set is unigrams with stemming. Let's now evaluate on the test set

[ ]:
```
X_train, y_train, X_test, y_test, X_dev, y_dev =␣
  ↪get_test_train_dev_split(stem_unigram_tf_idf)
multinomial_predictions = test_train_multinomial_bayes(X_train, y_train, X_test)
print("Multinomial Bayes")
evaluate_model(y_test, multinomial_predictions)
gaussian_predictions = test_train_gaussian_bayes(X_train, y_train, X_test)
print("Gaussian Bayes")
evaluate_model(y_test, gaussian_predictions)
test_train_sklearn_models(X_train, y_train, X_test, y_test)
```

```
Multinomial Bayes
accuracy: 0.80875
precision: 0.8768328445747801
recall: 0.7292682926829268
f1 score: 0.796271637816245

Gaussian Bayes
accuracy: 0.8275
precision: 0.8105022831050228
recall: 0.8658536585365854
f1 score: 0.8372641509433961

Sklearn Multinomial Bayes
```

```
accuracy: 0.82
precision: 0.8093023255813954
recall: 0.848780487804878
f1 score: 0.8285714285714286

Sklearn Gaussian Bayes
accuracy: 0.6525
precision: 0.6578947368421053
recall: 0.6707317073170732
f1 score: 0.6642512077294686
```

[ ]:

[ ]:

Logistic Regression

Let's first compare one hot matrices to our usual TF-IDF vectors

[ ]:
```
one_hot_matrix = generate_tf_idf_matrix(stemmed_unigrams, shared_vocabulary,␣
↪one_hot=True)
print(one_hot_matrix[0].toarray())
```

```
[[1. 1. 1. … 0. 0. 0.]]
```

[ ]:
```
from sklearn.linear_model import LogisticRegression
X_train, y_train, X_test, y_test, X_dev, y_dev =␣
↪get_test_train_dev_split(stem_unigram_tf_idf)
clf = LogisticRegression(random_state=r_seed, solver="sag")
clf.fit(X_train, y_train)
clf.score(X_dev, y_dev)
```

[ ]:
```
0.8104166666666667
```

[ ]:
```
X_train, y_train, X_test, y_test, X_dev, y_dev =␣
↪get_test_train_dev_split(one_hot_matrix)
clf = LogisticRegression(random_state=r_seed, solver="sag")
clf.fit(X_train, y_train)
clf.score(X_dev, y_dev)
```

```
c:\Users\adame\AppData\Local\Programs\Python\Python39\lib\site-
packages\sklearn\linear_model\_sag.py:350: ConvergenceWarning: The max_iter was
reached which means the coef_ did not converge
  warnings.warn(
```

[ ]:
```
0.8333333333333334
```

Clearly one hot matrices lead to higher performance. Let's try evaluating for stemmed unigrams:

[ ]:
```
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8333333333333334
precision: 0.8122270742358079
recall: 0.8340807174887892
f1 score: 0.8230088495575221
```

Let's try for lemmatized unigrams:

[ ]:
```
lem_uni_one_hot = generate_tf_idf_matrix(lemmatized_unigrams,
 ↪generate_shared_vocabulary(lemmatized_unigrams), one_hot=True)
X_train, y_train, X_test, y_test, X_dev, y_dev =␣
 ↪get_test_train_dev_split(lem_uni_one_hot)
clf = LogisticRegression(random_state=r_seed, solver="sag").fit(X_train,␣
 ↪y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8583333333333333
precision: 0.8414096916299559
recall: 0.8565022421524664
f1 score: 0.8488888888888888
```

```
c:\Users\adame\AppData\Local\Programs\Python\Python39\lib\site-
packages\sklearn\linear_model\_sag.py:350: ConvergenceWarning: The max_iter was
reached which means the coef_ did not converge
  warnings.warn(
```

Stemmed unigrams without stopword removal:

[ ]:
```
stem_uni_stop_one_hot = generate_tf_idf_matrix(stemmed_unigrams_stopwords,
 ↪generate_shared_vocabulary(stemmed_unigrams_stopwords), one_hot=True)
X_train, y_train, X_test, y_test, X_dev, y_dev =␣
 ↪get_test_train_dev_split(stem_uni_stop_one_hot)
clf = LogisticRegression(random_state=r_seed, solver="sag")
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8375
precision: 0.8111587982832618
recall: 0.8475336322869955
f1 score: 0.8289473684210525
```

```
c:\Users\adame\AppData\Local\Programs\Python\Python39\lib\site-
packages\sklearn\linear_model\_sag.py:350: ConvergenceWarning: The max_iter was
reached which means the coef_ did not converge
  warnings.warn(
```

The best performing feature set was thus lemmatization with stop-word removal.

SVMs

Stemmed unigrams with stopword removal

[ ]:
```
from sklearn import svm
def svm_classifier(feature_matrix):
    X_train, y_train, X_test, y_test, X_dev, y_dev =␣
 ↪get_test_train_dev_split(feature_matrix)
    clf = svm.SVC()
    clf.fit(X_train, y_train)
    predictions = clf.predict(X_dev)
    evaluate_model(y_dev, predictions)
svm_classifier(one_hot_matrix)
```

```
accuracy: 0.8520833333333333
precision: 0.8247863247863247
recall: 0.8654708520179372
f1 score: 0.8446389496717724
```

SVM with lemmatized unigrams

[ ]:
```
svm_classifier(lem_uni_one_hot)
```

```
accuracy: 0.85
precision: 0.8212765957446808
recall: 0.8654708520179372
f1 score: 0.8427947598253275
```

SVM with stemmed unigrams and no stopword removal

[ ]:
```
svm_classifier(stem_uni_stop_one_hot)
```

```
accuracy: 0.8520833333333333
precision: 0.8247863247863247
recall: 0.8654708520179372
f1 score: 0.8446389496717724
```

Our highest performing set was lemmatization with stopword removal. Let's now optimise hyper-parameters

Hyperparameter optimisation

Baseline performance with no hyperparameter tuning: accuracy: 0.8583333333333333 precision: 0.8414096916299559 recall: 0.8565022421524664 f1 score: 0.8488888888888888

Changing regularization parameter C:

[ ]:
```
#C=0.1
X_train, y_train, X_test, y_test, X_dev, y_dev =␣
 ↪get_test_train_dev_split(lem_uni_one_hot)
clf = LogisticRegression(C=0.1, random_state=r_seed, solver="sag")
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8541666666666666
precision: 0.84
recall: 0.8475336322869955
f1 score: 0.84375
```

[ ]:
```
#C=1.5
X_train, y_train, X_test, y_test, X_dev, y_dev =␣
 ↪get_test_train_dev_split(lem_uni_one_hot)
clf = LogisticRegression(C=1.5, random_state=r_seed, solver="sag")
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8583333333333333
precision: 0.8414096916299559
recall: 0.8565022421524664
f1 score: 0.8488888888888888


c:\Users\adame\AppData\Local\Programs\Python\Python39\lib\site-
packages\sklearn\linear_model\_sag.py:350: ConvergenceWarning: The max_iter was
reached which means the coef_ did not converge
  warnings.warn(
```

C=1.5 clearly improves performance. Now testing different solvers

[ ]:
```
#solver="liblinear"
clf = LogisticRegression(C=1.5, solver="liblinear",random_state=r_seed)
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.85625
precision: 0.8347826086956521
recall: 0.8609865470852018
f1 score: 0.847682119205298
```

[ ]:
```
#solver="lbfgs"
clf = LogisticRegression(C=1.5, solver="lbfgs", random_state=r_seed)
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8583333333333333
precision: 0.8384279475982532
recall: 0.8609865470852018
f1 score: 0.8495575221238938
```

[ ]:
```
#solver="sag"
clf = LogisticRegression(C=1.5, solver="sag",random_state=r_seed)
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8583333333333333
precision: 0.8414096916299559
recall: 0.8565022421524664
f1 score: 0.8488888888888888


c:\Users\adame\AppData\Local\Programs\Python\Python39\lib\site-
packages\sklearn\linear_model\_sag.py:350: ConvergenceWarning: The max_iter was
reached which means the coef_ did not converge
  warnings.warn(
```

"lbfgs" achieves the best performance. Now experimenting with different penalties

[ ]:
```
#penalty=None
clf = LogisticRegression(C=1.5, solver="lbfgs", random_state=r_seed,
                         penalty=None)
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
c:\Users\adame\AppData\Local\Programs\Python\Python39\lib\site-
packages\sklearn\linear_model\_logistic.py:1193: UserWarning: Setting
penalty=None will ignore the C and l1_ratio parameters
  warnings.warn(

accuracy: 0.8479166666666667
precision: 0.831858407079646
recall: 0.8430493273542601
f1 score: 0.8374164810690423
```

[ ]:
```
#penalty="l2"
clf = LogisticRegression(C=1.5, solver="lbfgs", random_state=r_seed,
                         penalty="l2")
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8583333333333333
precision: 0.8384279475982532
recall: 0.8609865470852018
f1 score: 0.8495575221238938
```

A penalty of "l2" appears optimal, now attempting to parallelize with n_jobs:

[ ]:
```
#n_jobs=-1
clf = LogisticRegression(C=1.5, solver="lbfgs", random_state=r_seed,
                         penalty="l2", n_jobs=-1)
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8583333333333333
precision: 0.8384279475982532
recall: 0.8609865470852018
f1 score: 0.8495575221238938
```

[ ]:
```
#n_jobs=None
clf = LogisticRegression(C=1.5, solver="lbfgs", random_state=r_seed,
                         penalty="l2", n_jobs=None)
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8583333333333333
precision: 0.8384279475982532
recall: 0.8609865470852018
f1 score: 0.8495575221238938
```

We find that parallelizing does not increase performance. Hyperparameter tuning is complete, giving us our fine tuned model: + C = 1.5 + solver = "lbfgs" + penalty = "l2" + n_jobs = None (i.e. not parallelized)

[ ]:
```
predictions = clf.predict(X_test)
evaluate_model(y_test, predictions)
```

```
accuracy: 0.83625
precision: 0.8329355608591885
```

```
recall: 0.8512195121951219
f1 score: 0.8419782870928829
```

SVM Hyperparameters

Baseline performance of untuned model: accuracy: 0.8520833333333333 precision: 0.8247863247863247 recall: 0.8654708520179372 f1 score: 0.8446389496717724

Testing different values for regulariation parameter C:

[ ]:
```
X_train, y_train, X_test, y_test, X_dev, y_dev =␣
↪get_test_train_dev_split(one_hot_matrix)
clf = svm.SVC(C=1.0)
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

[ ]:
```
clf = svm.SVC(C=0.9)
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8520833333333333
precision: 0.8220338983050848
recall: 0.8699551569506726
f1 score: 0.8453159041394336
```

C = 0.9 slightly outperforms the default C = 1, now testing different kernels

[ ]:
```
clf = svm.SVC(C=0.9, kernel="linear")
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

```
accuracy: 0.8166666666666667
precision: 0.8
recall: 0.8071748878923767
f1 score: 0.8035714285714287
```

[ ]:
```
clf = svm.SVC(C=0.9, kernel="rbf")
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

"rbf" kernel achieves the best perfomance, now testing different gamma configurations

[ ]:
```
clf = svm.SVC(C=0.9, kernel="rbf", gamma="auto")
clf.fit(X_train, y_train)
```

```
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

[ ]:
```
clf = svm.SVC(C=0.9, kernel="rbf", gamma="scale")
clf.fit(X_train, y_train)
predictions = clf.predict(X_dev)
evaluate_model(y_dev, predictions)
```

gamma="scale" appears to performe the best, giving us our final tuned model: + C = 0.9 + kernel = "rbf" + gamma = "scale"

[ ]:
```
predictions = clf.predict(X_test)
evaluate_model(y_test, predictions)
```

```
accuracy: 0.82625
precision: 0.8086560364464692
recall: 0.8658536585365854
f1 score: 0.8362779740871613
```

BERT Results

Our BERT experiments are contained in another notebook however we have included the results here for comparitive purposes.

[ ]:
```
"""
BERT-UNCASED: DEV SET
accuracy: 0.8458333333333333
precision: 0.8311111111111111
recall: 0.8385650224215246
f1 score: 0.8348214285714286

TEST SET
accuracy: 0.89125
precision: 0.8891566265060241
recall: 0.9
f1 score: 0.8945454545454546
"""
```

[ ]:
```
"""
BERT CASED DEV SET
accuracy: 0.84375
precision: 0.8032786885245902
recall: 0.8789237668161435
f1 score: 0.8394004282655245


TEST SET
accuracy: 0.8825
```

```
precision: 0.8657407407407407
recall: 0.9121951219512195
f1 score: 0.8883610451306413
"""
```

[ ]:

BERT_Experiments

December 12, 2023

BERT Experiments

[ ]:
```
import os
def read_corpus(directory):
    files = [f for f in os.listdir(directory) if f.endswith('.txt')]
    corpus = []
    for file in files:
        with open(os.path.join(directory, file), 'r', encoding='utf-8') as f:
            document = f.read()
            corpus.append(document)
    return corpus
```

[ ]:
```
#tested on google Colab due to resource limitations
from google.colab import drive
drive.mount('/content/drive')
positive_corpus = read_corpus("drive/MyDrive/nlp/data/pos/")
negative_corpus = read_corpus("drive/MyDrive/nlp/data/neg/")
corpus = positive_corpus + negative_corpus
positive_labels = len(positive_corpus)
negative_labels = len(negative_corpus)
corpus_length = len(corpus)

#sanity check
print(positive_corpus[0])
print(negative_corpus[1])
```

```
Mounted at /content/drive
I've long wanted to see this film, being a fan of both Peter Cushing and David
McCallum. I agree that the romantic sub-plot was a waste of time, but the talent
of McCallum shines through this juvie role. Thank heavens for Turner Classic
which aired the show last week. I can imagine that there were lots of problems
with children after the war, especially with the way things were throughout the
1950s. Some of the boys are a bit scary. I certainly wouldn't want to met them
on a well-lit street, much less a dark one. There were some good insights
regarding the feelings of a firebug as well, or as they call him, a firefly.
While this film certainly does possess the stench of a bad film, it's
surprisingly watchable on several levels. First, for old movie fans, it's
interesting to see the leading role played by Dean Jagger (no relation to Mick).
While Jagger later went on to a very respectable role as a supporting actor
```

(even garnering the Oscar in this category for 12 O'CLOCK HIGH), here his performance is truly unique since he actually has a full head of hair (I never saw him this way before) and because he was by far the worst actor in the film. This film just goes to show that if an actor cannot act in his earlier films doesn't mean he can't eventually learn to be a great actor. Another good example of this phenomenon is Paul Newman, whose first movie (THE SILVER CHALICE) is considered one of the worst films of the 1950s.

A second reason to watch the film is the shear cheesiness of it all. The writing is bad, the acting is bad and the special effects are bad. For example, when Jagger and an unnamed Cambodian are wading through the water, it's obvious they are really just walking in place and the background is poorly projected behind them. Plus, once they leave the water, their costumes are 100% dry!!! Horrid continuity and mindlessly bad dialog abounds throughout the film--so much so that it's hard to imagine why they didn't ask Bela Lugosi or George Zucco to star in the film--since both of them starred in many grade-z horror films. In many ways, this would be a perfect example for a film class on how NOT to make a film.

So, while giving it a 3 is probably a bit over-generous, it's fun to laugh at and short so it's worth a look for bad film fans.

[ ]:
```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.model_selection import train_test_split

r_seed = 563
np.random.seed(r_seed)

def get_test_train_dev_split(X):
    y = np.concatenate([np.ones(positive_labels), np.zeros(negative_labels)])

    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.80,
                                                        shuffle=True,␣
 ↪random_state=r_seed)

    X_train, X_dev, y_train, y_dev = train_test_split(X_train, y_train,
                                                      test_size=0.15,
                                                     shuffle=True,␣
 ↪random_state=r_seed)

    #68% train, 12% validation, 20% test
    return X_train, y_train, X_test, y_test, X_dev, y_dev
```

Training and testing the BERT models

REFERENCES [1] https://huggingface.co/transformers/v3.2.0/custom_datasets.html [2] https://huggingface.co/docs/transformers/tasks/sequence_classification Note that boilerplate code was directly taken from the above sites for the loading of our own dataset and the intialisation of the BERT models. Experiments were also made using the uncased model, as demonstrated through the commented lines of code.

[ ]:
```
! pip install -U accelerate
! pip install -U transformers
```

```
Collecting accelerate
  Downloading accelerate-0.25.0-py3-none-any.whl (265 kB)
                             265.7/265.7
kB 3.0 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17 in
/usr/local/lib/python3.10/dist-packages (from accelerate) (1.23.5)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from accelerate) (23.2)
Requirement already satisfied: psutil in /usr/local/lib/python3.10/dist-packages
(from accelerate) (5.9.5)
Requirement already satisfied: pyyaml in /usr/local/lib/python3.10/dist-packages
(from accelerate) (6.0.1)
Requirement already satisfied: torch>=1.10.0 in /usr/local/lib/python3.10/dist-
packages (from accelerate) (2.1.0+cu118)
Requirement already satisfied: huggingface-hub in
/usr/local/lib/python3.10/dist-packages (from accelerate) (0.19.4)
Requirement already satisfied: safetensors>=0.3.1 in
/usr/local/lib/python3.10/dist-packages (from accelerate) (0.4.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from torch>=1.10.0->accelerate) (3.13.1)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.10/dist-packages (from torch>=1.10.0->accelerate) (4.5.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch>=1.10.0->accelerate) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-
packages (from torch>=1.10.0->accelerate) (3.2.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch>=1.10.0->accelerate) (3.1.2)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch>=1.10.0->accelerate) (2023.6.0)
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-
packages (from torch>=1.10.0->accelerate) (2.1.0)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from huggingface-hub->accelerate) (2.31.0)
Requirement already satisfied: tqdm>=4.42.1 in /usr/local/lib/python3.10/dist-
packages (from huggingface-hub->accelerate) (4.66.1)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch>=1.10.0->accelerate)
(2.1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->huggingface-
hub->accelerate) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->huggingface-hub->accelerate) (3.6)
```

```
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->huggingface-
hub->accelerate) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->huggingface-
hub->accelerate) (2023.11.17)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-
packages (from sympy->torch>=1.10.0->accelerate) (1.3.0)
Installing collected packages: accelerate
Successfully installed accelerate-0.25.0
Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-
packages (4.35.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from transformers) (3.13.1)
Requirement already satisfied: huggingface-hub<1.0,>=0.16.4 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.19.4)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-
packages (from transformers) (1.23.5)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from transformers) (23.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-
packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from transformers) (2023.6.3)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from transformers) (2.31.0)
Requirement already satisfied: tokenizers<0.19,>=0.14 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.15.0)
Requirement already satisfied: safetensors>=0.3.1 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.4.1)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-
packages (from transformers) (4.66.1)
Requirement already satisfied: fsspec>=2023.5.0 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.16.4->transformers) (2023.6.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.16.4->transformers) (4.5.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->transformers) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers)
(2023.11.17)
```

[ ]:
```
X_train, y_train, X_test, y_test, X_dev, y_dev =␣
↪get_test_train_dev_split(corpus)
```

[ ]:
```
import torch

#tokenise and encode the dataset:
from transformers import DistilBertTokenizerFast
#tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-cased')
train_encodings = tokenizer(X_train, truncation=True, padding=True)
val_encodings = tokenizer(X_dev, truncation=True, padding=True)
test_encodings = tokenizer(X_test, truncation=True, padding=True)
```

```
tokenizer_config.json:   0%|          | 0.00/29.0 [00:00<?, ?B/s]
vocab.txt:   0%|          | 0.00/213k [00:00<?, ?B/s]
tokenizer.json:   0%|          | 0.00/436k [00:00<?, ?B/s]
config.json:   0%|          | 0.00/465 [00:00<?, ?B/s]
```

[ ]:
```
#loads the set into the class wrapper for use by BERT
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]).long()#.unsqueeze(1)
        #NOTE that the .long() is necessary for for functionality,
        #this line was added during development
        return item

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

train_dataset = IMDbDataset(train_encodings, y_train)
val_dataset = IMDbDataset(val_encodings, y_dev)
test_dataset = IMDbDataset(test_encodings, y_test)
```

[ ]:
```
train_dataset[1]
```

[ ]:
```
{'input_ids': tensor([  101,  7595,   146,  1189,   170,  6223,  1105,   146,
3004,   128,
         11854,  1116,  1120,  1103,  2523,  5184,  1106,  2824,  1142,  8327,
          2764,  2008,  2523,   119,  1422,  1827,   132,   133,  9304,   120,
           135,   133,  9304,   120,   135,  2352,  1110,  1359,  1113,   123,
```

```
 1614,   132,   133,  9304,   120,   135,   133,  9304,   120,   135,
  122,   114, 27673,  1348,  1553,  1104, 10344,   131,  1249,  1122,
 5940,  1113,  1211,  1104,  1103,  1237,  9141,   117,  1103,  2432,
 6191,  2111,  1112,  1126,  6640,  1170,  3776,   123,  1137,   124,
 1614,  1164,  1103,  3141,  2754,   119,  1252, 16679,  1122,  1110,
 1136,  1536,   119, 14881,   180, 10038,  8124,  1105,   123,  2666,
 1104,  1168, 11785,  2144,   112,   189,  1294,   170,  1825,  2437,
  170,  2754,   119,  1370,  1859,   188, 22300,  1110,  1103,  2951,
 1104,  4044,  1297,  1107,  3165,  1133,  2490,  1108,  1774,  1103,
 1873,  1106,  1831,  1115,   119, 12772,  1234,  1132,  2131,   113,
23330,  1152,  1400,  1115,   114,  1105,  1147,  3057,  2769,  1105,
 8810,  1105,  8015,  1132,  2423,  1472,  1121,  1168,  4944,  2182,
  119,  1109,  1514,  3719,  1206,  2638,  1105,  2466,  2754,  1110,
 1195,  1274,   112,   189,  1294,  3084,  7272, 13335,  3452, 18502,
  119,  1573,  1195,  1274,   112,   189,  3942,  1234,  1170,  1147,
 1148,  2304,  1164,  1412,  1297,  1112,  1103,  1401,  2482,  1225,
 1107,  1155,  1104,  1103,  1273,   119,   133,  9304,   120,   135,
  133,  9304,   120,   135,   123,   114, 21843,  8011,  1104,   170,
 1873,   131,  1247,  1110,  1720,  1277,  1106,  1474,  1164,  1142,
  119,  3237,  1143,  1275,  2636,  1134,  1125,  1292,  1113,  1147,
 3785,  8264,  3381,  1190,   146,  1209,  1474,  1115,   146,  1821,
 2488,   119,   133,  9304,   120,   135,   133,  9304,   120,   135,
  146,  1724,  1142,  7368,  1272,  1103,  6419,  1132,  7495,  1103,
 1273,  1107,  1103,  1602,  8594,  6453,   119,  4203,  2824,  1987,
  119, 11359, 24040,  1105,  2437,  1103,  2764,  1104,  1602,  8594,
  119,   138,  1602,  8594,  1144,  1106,  7977,  1103,  3062,  1105,
 1144,  1106,  2817,  1103,  3703,  1106,  1103,  6276,  2192,  1104,
 1122,   119,  2777,  1110,  1103,  3062,   136,  2777,  1110,  1103,
 2764,  1164,  1103,  2523,   119,   102,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
    0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
```

```
          0,      0]),
 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
         1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0]),
 'labels': tensor(0)}
```

[ ]:
```
id2label = {0: "NEGATIVE", 1: "POSITIVE"}
label2id = {"NEGATIVE": 0, "POSITIVE": 1}
```

[ ]:
```
import accelerate
import transformers
from transformers import DistilBertForSequenceClassification, Trainer,␣
 ↪TrainingArguments

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 16
    per_device_eval_batch_size=64,   # batch size for evaluation 64
    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")
model = DistilBertForSequenceClassification.
 ↪from_pretrained("distilbert-base-cased")

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

trainer.train()
```

```
model.safetensors:   0%|          | 0.00/263M [00:00<?, ?B/s]
```

```
Some weights of DistilBertForSequenceClassification were not initialized from
the model checkpoint at distilbert-base-cased and are newly initialized:
['classifier.bias', 'classifier.weight', 'pre_classifier.bias',
'pre_classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

```
<IPython.core.display.HTML object>
```

[ ]:
```
TrainOutput(global_step=510, training_loss=0.3883705898827198,
metrics={'train_runtime': 374.1151, 'train_samples_per_second': 21.811,
'train_steps_per_second': 1.363, 'total_flos': 1080933973032960.0, 'train_loss':
0.3883705898827198, 'epoch': 3.0})
```

[ ]:
```
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score

def evaluate_model( test_labels, predictions):
    print("accuracy:", accuracy_score(test_labels, predictions))
    print("precision:", precision_score(test_labels, predictions))
    print("recall:", recall_score(test_labels, predictions))
    print("f1 score:", f1_score(test_labels, predictions))
    print()
```

[ ]:
```
#development set evaluation
predictions = trainer.predict(val_dataset)
predicted_classes = predictions.predictions.argmax(axis=1)
evaluate_model(y_dev, predicted_classes)
```

```
<IPython.core.display.HTML object>
```

```
accuracy: 0.84375
precision: 0.8032786885245902
recall: 0.8789237668161435
f1 score: 0.8394004282655245
```

[ ]:
```
#test set evaluation
predictions = trainer.predict(test_dataset)
predicted_classes = predictions.predictions.argmax(axis=1)
evaluate_model(y_test, predicted_classes)
```

```
<IPython.core.display.HTML object>
```

```
accuracy: 0.8825
precision: 0.8657407407407407
recall: 0.9121951219512195
f1 score: 0.8883610451306413
```

[ ]:

[ ]: