An Introduction into implementing Sentiment Analysis using Naïve Bayes and BERT

Contents

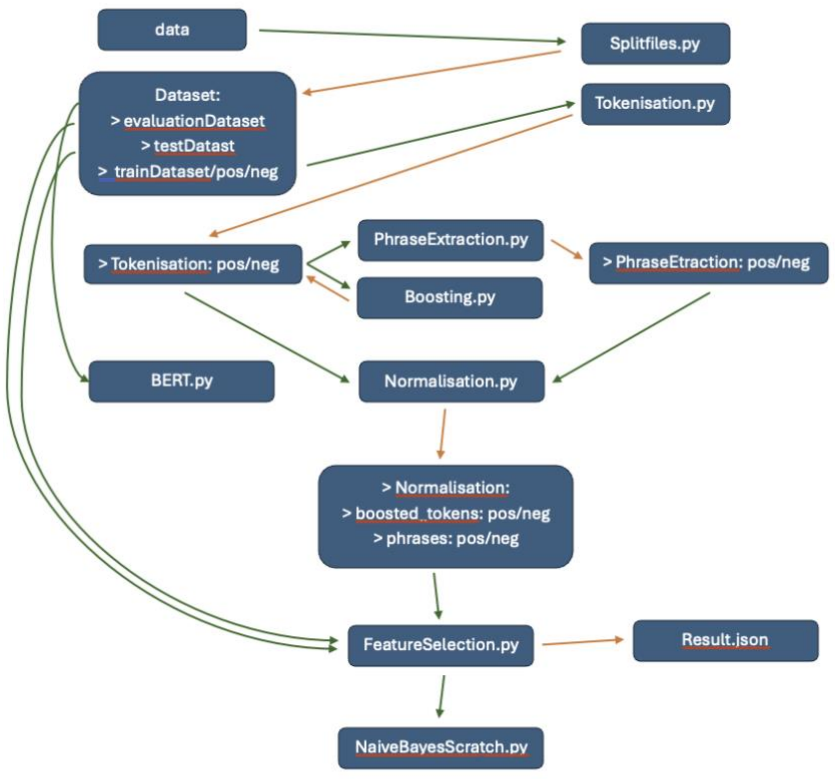
1.0 Introduction

Sentiment Analysis is a sub-field of Natural Language Processing the focuses on identifying and categorizing opinions in each text, with its main aim to identify its sentiment polarity whether it conveys a positive, negative, or neutral sentiment. It's been widely applied in industries like marketing, customer service, and social media monitoring. This in turn enables businesses to gauge public opinion, enhance customer experiences, and make data-driven decisions.
In this project I was tasked with training a Naïve Bayes Model to detect a positive from a negative movie review based on the principles of Sentiment Analysis. My implementation includes separate Python files for each step making it easier to modify each individual functionality individually making it easier for troubleshooting possible issue, whilst also storing the output at each step in .json files that would be read as input/reference for the following functions.

1.1 File Names with short descriptions:

1. main.py: Structure that sequences all the functionality
2. Splitfiles.py: Used to split the provided review data into train, evaluation and test folders.
3. Tokenisation.py: Tokenizes each review through Lemmatization, Stemming and splitting against all tokens and white spaces; Also ensures token dropping through frequency normalization
4. PhraseExtraction.py: Extracts n-grams (bigram and trigram) along with noun phrases
5. Boosting.py: External feature that boosts/increases the number of occurrences of said word in a review by a fraction, by checking if synonyms of that token exist.
6. Normalisation.py: Used Binary and TF-IDF to normalise the data we processed after phrase extraction and token boosting.
7. FeatureSelection.py: Implemented Naïve Bayes to train on the normalise data to predict positive from negative reviews; Find the best 'feature set' for training.
8. NaiveBayesScratch.py: Coded a Naïve Bayes from the ground up to predict the sentiment polarity of reviews based on the best 'feature set' gotten from the 'Feature Selection'.
9. FileReader.py: Created helper file to read, save to and categorise .json files
10. BERT.py: Implemented BERT

1.2 Workflow of the main.py:



Green lines: Input
Orange lines: Output

We start from data and make our way down. The green arrows indicate the input for each python file while the orange arrows indicate the output for the following files

2.0 Evaluation Data Splits:

- Given dataset contained 2000 positive and 2000 negative reviews in 2 subfolders 'pos' and 'neg' respectively.
- Copied both sub-folders into new folder 'trainDataset'
- Split the data as a 70%, 15%, 15% split amongst the 'trainDataset', 'evaluationDataset' and 'testDataset'
- The 'trainDataset' consisted of 2 subfolders 'pos' and 'neg' to separate the positive from negative reviews; While the 'evaluationDataset' and 'testDataset' consisted of a shuffled collection of mixed reviews with the substring 'pos_' and 'neg_' prefixed to each respective file name respectively. This was done so that we could calculate the accuracy, precision and F1 score of the trained model on a mixed collection of reviews.
- We require a large amount of data to train on hence why I have chosen to train on 70% of data, giving it enough examples to capture complex relationships between features and target labels. It also reduced the risk of underfitting.
- 'evaluationDataset' is used to tune the model parameters, select feature and compare model variants without biasing the final test result, hence why it is set at 15% of the given data.
- 'testDataset' contains the remaining 15% on which we can test out trained model. The allocation strikes a balance between having enough data for a robust evaluation and leaving most of the data for training.

3.0 Feature Generation, Dimensionality Reduction and Feature Selection

3.1 Tokenisation:

- Split each review into a selection of tokens through 3 main methods '**White Space Elimination (include all tokens)', 'Lemmatization' and 'Stemming'**.
- **'Stop Words'** are common words used that hold no value in providing any sentiment analysis. These words are common conjunctions, articles, pronouns, prepositions, auxiliary verbs, etc. I imported the 'Stop Words' from the 'nltk.corpus' library and set it to English words. I implemented not including Stop Words in all my Tokenization methods for uniformity as well as converting all words to lowercase.
- **'White Space Elimination'**: This results in a list containing only words separated with a space, other non-alpha-numeric character or tags. I used a Regular Expression to split the words up into a list.
- **'Stemming'**: It is the process of reducing a word to its root or base form, known as a *stem*. The stem may not always be a valid word but represents the word's core meaning. Stemming helps to group together different forms of a word so that they can be analysed as a single item.
  Eg: "running," "runs," "runner" are reduced to the stem "run."
- Stemming was implemented by calling the inbuilt algorithm 'PorterStemmer' from 'nltk.stem'
- <u>Advantages of Stemming:</u>
  - *Reduced Dimensionality:* By collapsing words with similar meanings into a single stem, stemming reduces the number of unique tokens.
  - *Improved Matching:* Variations of the same word are treated as equivalent, which can enhance text matching and retrieval tasks.
  - *Simplified Representation:* Stems focus on the core meaning, ignoring variations that may not affect the analysis.
- <u>Disadvantages of Stemming:</u>
  - *Loss of Precision:* Stemming can sometimes over-simplify words, leading to unrelated words being grouped together (e.g., "universal" and "universe" → "univers").
  - *Not Context-Aware*: Stemming does not consider the context in which a word is used, which might lead to inaccurate reductions.
- **'Lemmatization'**: It also reduces a word to its root or base form, but unlike stemming, it ensures that the resulting word is a *valid word in the language*. Lemmatization uses linguistic rules and considers the word's part of speech (PoS) for accurate transformation.
  Eg: "running," "runs," "ran" are reduced to "run."
  "better" are reduced to "good" (considering comparative forms).
- Lemmatization was implemented by calling the inbuilt algorithm 'WordNetLemmatizer' from 'nltk.stem'
- <u>Advantages of Lemmatization:</u>
  - *Linguistic Accuracy*: Lemmatization produces meaningful words, unlike stemming, which may yield truncated words
  - *Context-Awareness:* By considering the part of speech, lemmatization provides a more precise normalization (e.g., "running" → "run" for verbs, but "running" → unchanged if used as a noun like "the running").
  - *Improved Model Performance:* In tasks like text classification or sentiment analysis, lemmatization helps reduce dimensionality while retaining semantic meaning.

- Disadvantages of Lemmatization:
    - o *Computationally Intensive:* Lemmatization is more complex and slower than stemming as it requires morphological analysis and PoS tagging.
    - o *Dependency on PoS Tagging:* Errors in PoS tagging can lead to incorrect lemmas (e.g., treating "running" as a noun instead of a verb).
- To reduce the number of Tokens we use I decided to include a frequency normalization function that counts all the occurrences of a particular word in a review and gauges its frequency. If its frequency exceeds the threshold frequency, in my implementation that would be 5, we will remove it from the list of tokenized words for that review.
- Out output would consist of 6 files; 3 files using all 3 tokenization's for positive reviews and saved it to the 'pos' folder in Tokenisation and the 3 tokenized files for negative reviews saved in the 'neg' folder.

3.2 Compositional Phrases

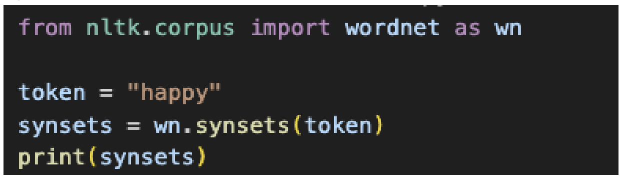
N-Grams: This includes breaking down text into contiguous sequence of n-tokens this could include words and characters. The idea of this being able to capture context, co-occurrence patterns and structure within the text.

- In my project I have implemented Bigrams, Trigrams and PoS + constituency parsing for noun phrases.
- For this task I took the input from the 'Tokenisation' folder. I implemented Bigrams and Trigrams through the pre-existing functionalities 'BigramCollocationFinder' and 'TrigramCollocationFinder' respectively from the 'nltk.collocations' library.
- I passed each review from the list of tokenized reviews individually to the respective N-gram function that first converted the single tokenized words into a '2-word' and '3-word' token separated by a space into its respective bigram and trigram list.
- I then joined the list of Bigrams and Trigrams to the pre-existing list of tokenized words separately for each review individually.
- Eg:
  Tokenized Lemmatized List: (contains 1 review)
  ['happy','car','race','excitement','senna','rain','wind','storm']
  Final Bigram List:
  ['happy', 'car', 'race', 'excitement', 'senna', 'rain', 'wind', 'storm', 'happy car', 'car race', 'race excitement', … , 'wind storm']
  Final Trigram List:
  ['happy','car','race','excitement','senna','rain','wind','storm', 'happy car race', 'car race excitement ', … , 'rain wind storm']
- I would then assign the occurrence of each phrase/token in each individual review to the new tokens using the Counter() to now convert it into a list of dictionaries with the frequency of each token per review.

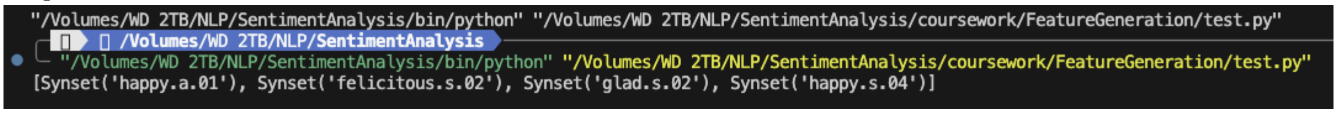
PoS + constituency parsing for noun phrases: The goal is to identify contiguous sequences of words that form noun phrases. A noun phrase typically contains a noun as the head word, optionally preceded or followed by determiners, adjectives, or other modifiers**.**

- For the input we used the tokenized List we generated. Subsequently we check if a noun tag is encountered, the token is added to the current noun phrase. We can check if a noun tag is encountered using the 'NN'/ 'NNS' (single/plural noun), 'NNP' or even 'NNPS'.
- To assign part-of-speech tags to tokens we can use NLTK's 'pos_tag' function.
- If on the chance a non-noun tag is encountered, the current phrase is completed and is then stored, and it continues with the remaining tokens. We also return the frequency of the token.
- Eg:
  Input: A list of tokens, ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"].
  PoS Tagging: Assigns PoS tags to tokens: [("The", "DT"), ("quick", "JJ"), ("brown", "JJ"), ("fox", "NN"), ...].
  Noun Phrase Detection:
  If the tag is NN, add the word to current_phrase.
  If the tag is not NN, finalize the current phrase, store it, and reset current_phrase.
  Output: Returns a frequency dictionary of noun phrases, e.g., {"brown fox": 1, "lazy dog": 1}.
- At the end we return a collection of files with the respective bigram, trigrams and noun phrases for each individual tokenization version.

3.3 Boosting Feature

- In this feature implement we 'boost'/increase the frequency of a token by a small value if a 'weak synonym' of that word exists also called, '*Hypernyms*'.
- Using the collection of Tokenized words as the inputs. To identify the possible 'hypernyms' of a word/token I have taken the help of synsets() a function that belongs to the wordnet class of the 'nltk.corpus' library. This function returns all the hypernyms of a particular word as a list.
  Eg:
  

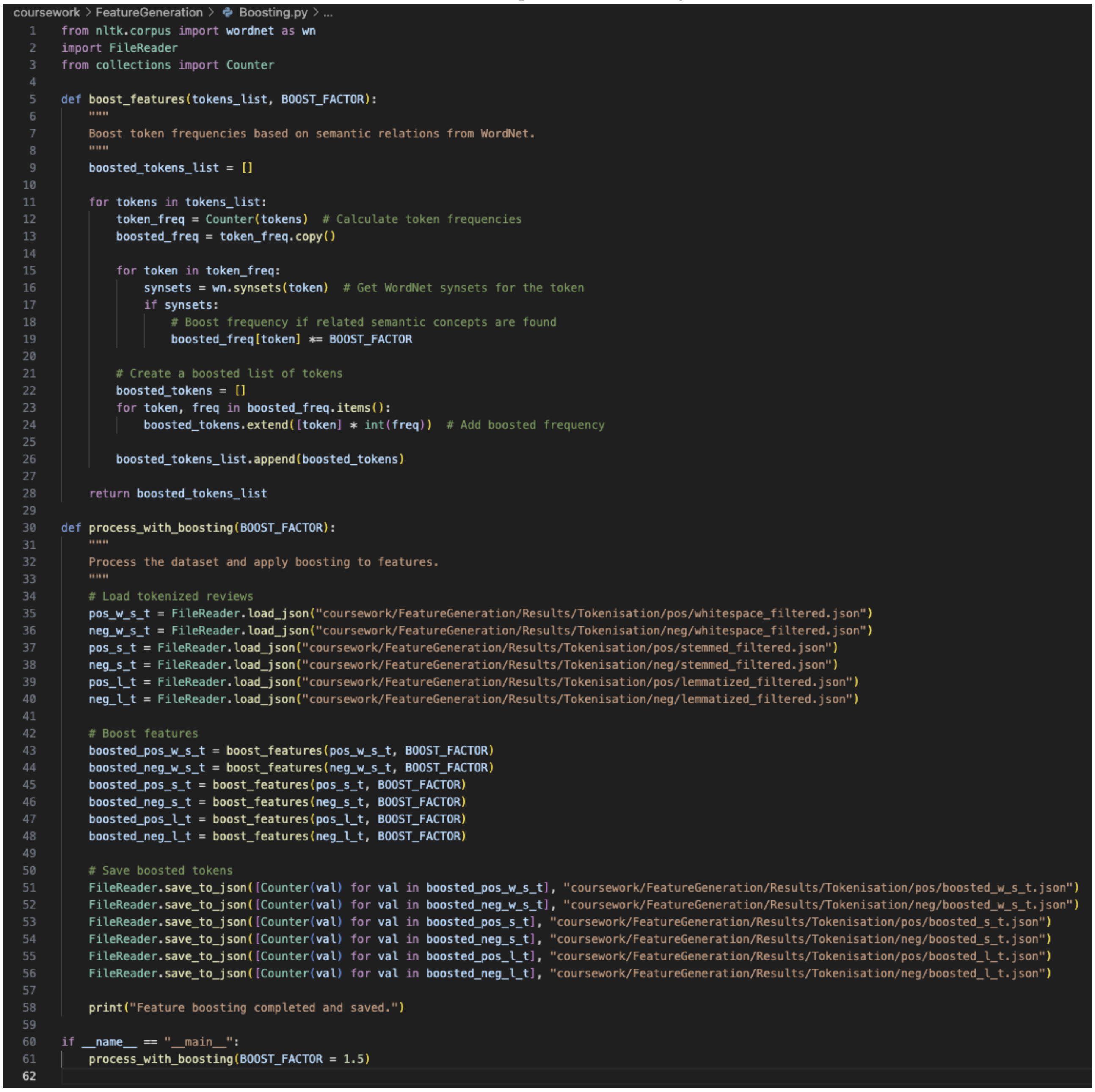
  ```
  from nltk.corpus import wordnet as wn

  token = "happy"
  synsets = wn.synsets(token)
  print(synsets)
  ```

  Output:
  

  ```
   "/Volumes/WD 2TB/NLP/SentimentAnalysis/bin/python" "/Volumes/WD 2TB/NLP/SentimentAnalysis/coursework/FeatureGeneration/test.py"
  ╭  /Volumes/WD 2TB/NLP/SentimentAnalysis
  ╰ "/Volumes/WD 2TB/NLP/SentimentAnalysis/bin/python" "/Volumes/WD 2TB/NLP/SentimentAnalysis/coursework/FeatureGeneration/test.py"
   [Synset('happy.a.01'), Synset('felicitous.s.02'), Synset('glad.s.02'), Synset('happy.s.04')]
  ```

- With this I was able to identify the tokens which were hypernyms and increase the frequency count value by multiplying it with a 'BOOST_FACTOR'. In this version of the code, I chose 1.5, as I saw this to be the best value that showed a significant change in value.
- After increasing the count value I created a new 'boost_tokens[]' list and appended the token with the updated count value.
- I then update the 'Tokenisation' folder with this as 'boosted_...' values that could be normalised and then used as training data.
- <u>What is the benefit to using Boosting?</u>
  - Improve Feature Importance
  - Enhances Common Contextual Understanding
  - Reduced Sparsity
- <u>Why I chose my specific Boosting method</u>
  - Leveraging Semantic Hierarchies: In simple words, hypernyms allow for us to represent more general concepts that encapsulate a word's meaning. Eg: 'car' could be linked to 'vehicle', this relation captures both the specific and broader context of words.
  - Controlled Impact through Multiplicative Factor: The boosting factor can account for meaningful adjustment, hence balancing the importance of boosted features without disproportionately dominating other fields.
  - Efficient: Utilising pre-existing dictionary of hypernyms through 'WordNet' 'synset'.
  - Integration with TF-IDF: Since we are actively boosting relevant words, Normalisation techniques such as TF-IDF could benefit with the increased frequency of relevant tokens.
  - Increases Coverage: Hypernyms are better used that Synonyms as they not only align better with the core meaning of the token but in doing so does without Overgeneralization.

- Code that implements Boosting:



coursework > FeatureGeneration > Boosting.py > ...

```python
from nltk.corpus import wordnet as wn
import FileReader
from collections import Counter

def boost_features(tokens_list, BOOST_FACTOR):
    """
    Boost token frequencies based on semantic relations from WordNet.
    """
    boosted_tokens_list = []

    for tokens in tokens_list:
        token_freq = Counter(tokens)  # Calculate token frequencies
        boosted_freq = token_freq.copy()

        for token in token_freq:
            synsets = wn.synsets(token)  # Get WordNet synsets for the token
            if synsets:
                # Boost frequency if related semantic concepts are found
                boosted_freq[token] *= BOOST_FACTOR

        # Create a boosted list of tokens
        boosted_tokens = []
        for token, freq in boosted_freq.items():
            boosted_tokens.extend([token] * int(freq))  # Add boosted frequency

        boosted_tokens_list.append(boosted_tokens)

    return boosted_tokens_list

def process_with_boosting(BOOST_FACTOR):
    """
    Process the dataset and apply boosting to features.
    """
    # Load tokenized reviews
    pos_w_s_t = FileReader.load_json("coursework/FeatureGeneration/Results/Tokenisation/pos/whitespace_filtered.json")
    neg_w_s_t = FileReader.load_json("coursework/FeatureGeneration/Results/Tokenisation/neg/whitespace_filtered.json")
    pos_s_t = FileReader.load_json("coursework/FeatureGeneration/Results/Tokenisation/pos/stemmed_filtered.json")
    neg_s_t = FileReader.load_json("coursework/FeatureGeneration/Results/Tokenisation/neg/stemmed_filtered.json")
    pos_l_t = FileReader.load_json("coursework/FeatureGeneration/Results/Tokenisation/pos/lemmatized_filtered.json")
    neg_l_t = FileReader.load_json("coursework/FeatureGeneration/Results/Tokenisation/neg/lemmatized_filtered.json")

    # Boost features
    boosted_pos_w_s_t = boost_features(pos_w_s_t, BOOST_FACTOR)
    boosted_neg_w_s_t = boost_features(neg_w_s_t, BOOST_FACTOR)
    boosted_pos_s_t = boost_features(pos_s_t, BOOST_FACTOR)
    boosted_neg_s_t = boost_features(neg_s_t, BOOST_FACTOR)
    boosted_pos_l_t = boost_features(pos_l_t, BOOST_FACTOR)
    boosted_neg_l_t = boost_features(neg_l_t, BOOST_FACTOR)

    # Save boosted tokens
    FileReader.save_to_json([Counter(val) for val in boosted_pos_w_s_t], "coursework/FeatureGeneration/Results/Tokenisation/pos/boosted_w_s_t.json")
    FileReader.save_to_json([Counter(val) for val in boosted_neg_w_s_t], "coursework/FeatureGeneration/Results/Tokenisation/neg/boosted_w_s_t.json")
    FileReader.save_to_json([Counter(val) for val in boosted_pos_s_t], "coursework/FeatureGeneration/Results/Tokenisation/pos/boosted_s_t.json")
    FileReader.save_to_json([Counter(val) for val in boosted_neg_s_t], "coursework/FeatureGeneration/Results/Tokenisation/neg/boosted_s_t.json")
    FileReader.save_to_json([Counter(val) for val in boosted_pos_l_t], "coursework/FeatureGeneration/Results/Tokenisation/pos/boosted_l_t.json")
    FileReader.save_to_json([Counter(val) for val in boosted_neg_l_t], "coursework/FeatureGeneration/Results/Tokenisation/neg/boosted_l_t.json")

    print("Feature boosting completed and saved.")

if __name__ == "__main__":
    process_with_boosting(BOOST_FACTOR = 1.5)
```

4.0 Normalise

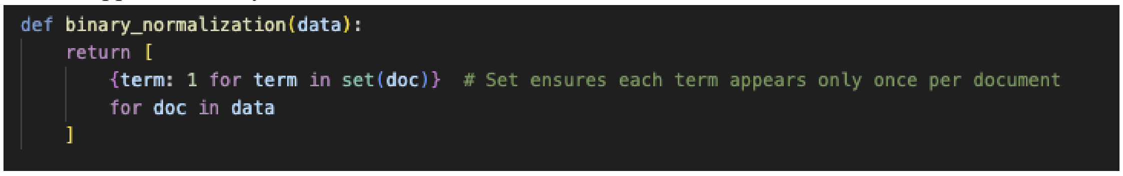
To be able to train the Naïve Bayes model we need to normalise the results we have gotten from boosting as well as from phrase extraction.
Apart from using TF-IDF I have opted to use Binary Normalization as the other method. Other options included using Term Frequency Normalisation, Frequency Normalisation, L2 Normalisation, etc.

4.1 Binary Normalization:

The reason why I picked Binary Normalization:

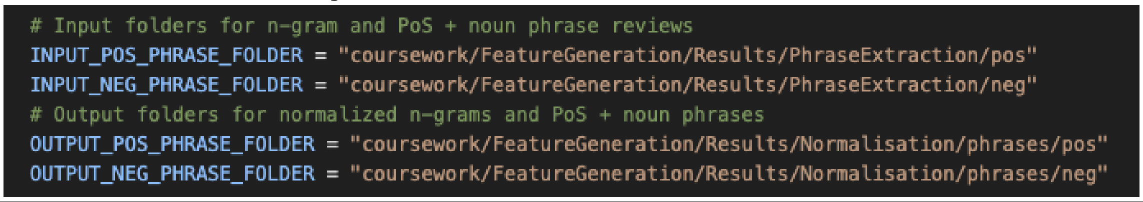
1. **Simplicity and Robustness:**
   - Binary normalization is simpler and focuses solely on whether a phrase exists in a document, reducing the complexity of feature vectors.
   - It avoids the risks of overfitting caused by overly frequent phrases dominating the model.
2. **Handling Boosted Features:**
   - *Task Focus:* If the task depends primarily on whether a term exists (e.g., detecting the presence of specific phrases in sentiment analysis), boosting complements binary normalization by ensuring boosted features are included in the presence/absence matrix.
   - *Counteracting Frequency Inflation*: When boosting heavily inflates feature counts, binary normalization prevents the model from being overwhelmed by these inflated values, ensuring balance among features.
   - Example: If "great" is boosted due to synonyms, its mere presence, not its inflated count, matters for sentiment classification.
3. **Reduced Overfitting:**
   - Models trained on frequency-based normalization (TF, TF-IDF) can be overfit to certain frequent phrases or phrases that appear in many documents.
   - Binary normalization generalizes better by ignoring frequency variations that may be dataset-specific noise.
4. **Focus on Phrase Presence:**
   - For phrase extraction (e.g., bigrams or noun phrases), the presence of specific meaningful phrases (e.g., "hidden gem," "poor acting") often matters more than how many times they appear.
5. **Interpretable Features:**
   - Binary phrases or tokens in sentiment prediction.

- Code Snippet for Binary Normalisation:

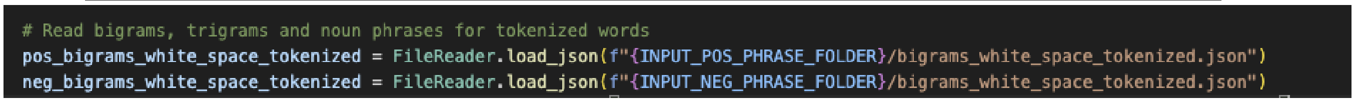


```python
def binary_normalization(data):
    return [
        {term: 1 for term in set(doc)}  # Set ensures each term appears only once per document
        for doc in data
    ]
```

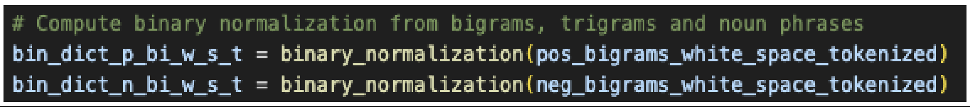
- Example of how we call and save the output:



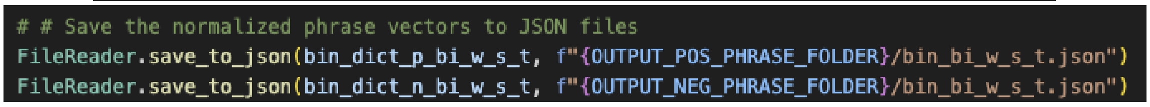
```python
# Input folders for n-gram and PoS + noun phrase reviews
INPUT_POS_PHRASE_FOLDER = "coursework/FeatureGeneration/Results/PhraseExtraction/pos"
INPUT_NEG_PHRASE_FOLDER = "coursework/FeatureGeneration/Results/PhraseExtraction/neg"
# Output folders for normalized n-grams and PoS + noun phrases
OUTPUT_POS_PHRASE_FOLDER = "coursework/FeatureGeneration/Results/Normalisation/phrases/pos"
OUTPUT_NEG_PHRASE_FOLDER = "coursework/FeatureGeneration/Results/Normalisation/phrases/neg"
```



```python
# Read bigrams, trigrams and noun phrases for tokenized words
pos_bigrams_white_space_tokenized = FileReader.load_json(f"{INPUT_POS_PHRASE_FOLDER}/bigrams_white_space_tokenized.json")
neg_bigrams_white_space_tokenized = FileReader.load_json(f"{INPUT_NEG_PHRASE_FOLDER}/bigrams_white_space_tokenized.json")
```



```python
# Compute binary normalization from bigrams, trigrams and noun phrases
bin_dict_p_bi_w_s_t = binary_normalization(pos_bigrams_white_space_tokenized)
bin_dict_n_bi_w_s_t = binary_normalization(neg_bigrams_white_space_tokenized)
```



```python
# # Save the normalized phrase vectors to JSON files
FileReader.save_to_json(bin_dict_p_bi_w_s_t, f"{OUTPUT_POS_PHRASE_FOLDER}/bin_bi_w_s_t.json")
FileReader.save_to_json(bin_dict_n_bi_w_s_t, f"{OUTPUT_NEG_PHRASE_FOLDER}/bin_bi_w_s_t.json")
```

4.2 TF-IDF

In this implementation we look at utilizing TF-IDF (Term Frequency – Inverse Document Frequency) to normalise the input tokens.

- It mainly consists of calculating the 'TF' part and the 'IDF' separately and then multiplying them together
- TF shows the relevance of the term in the document, while IDF ensures that common words are downweighed.
- *TF* is calculated as:

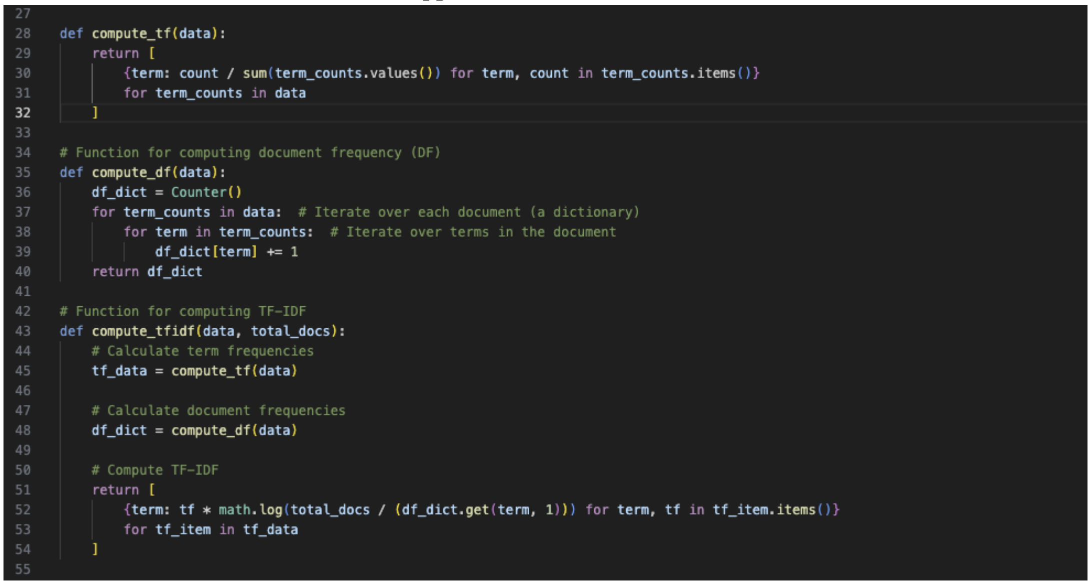
$$TF(t,d) = \frac{\text{Number of occurrences of } t \text{ in } d}{\text{Total number of terms in } d}$$

IDF is calculated as:

$$IDF(t) = \log \frac{\text{Total number of documents}}{\text{Number of documents containing } t}$$

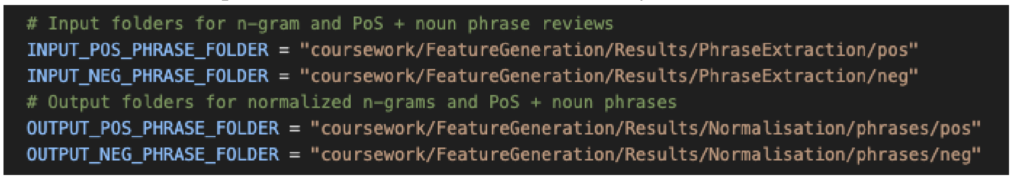
- The result includes multiplying TF with the IDF value = TF x IDF
- Reason why we use it:
  Feature Weighting:
  - Words like "the," "and," or "is" (common stopwords) have high term frequency but low IDF scores, reducing their overall importance.
  - Important and unique terms get higher TF-IDF scores.
  Dimensionality Reduction:
  - By focusing on terms with high TF-IDF scores, we can filter out less informative terms, reducing computational overhead in machine learning models.
  Improved Relevance:
  - It emphasizes distinctive words, improving the performance of tasks like text classification and information retrieval.

Code snippet for TF-IDF Normalisation:

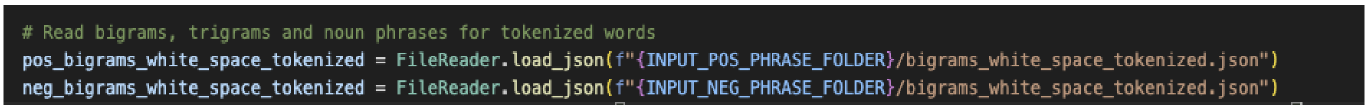


```python

def compute_tf(data):
    return [
        {term: count / sum(term_counts.values()) for term, count in term_counts.items()}
        for term_counts in data
    ]

# Function for computing document frequency (DF)
def compute_df(data):
    df_dict = Counter()
    for term_counts in data:  # Iterate over each document (a dictionary)
        for term in term_counts:  # Iterate over terms in the document
            df_dict[term] += 1
    return df_dict

# Function for computing TF-IDF
def compute_tfidf(data, total_docs):
    # Calculate term frequencies
    tf_data = compute_tf(data)

    # Calculate document frequencies
    df_dict = compute_df(data)

    # Compute TF-IDF
    return [
        {term: tf * math.log(total_docs / (df_dict.get(term, 1))) for term, tf in tf_item.items()}
        for tf_item in tf_data
    ]

```

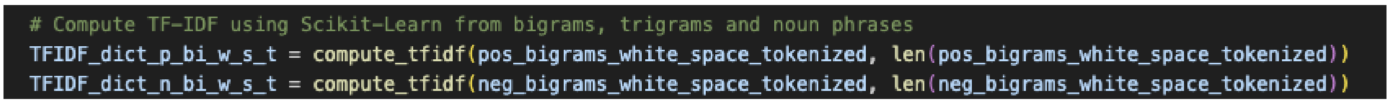
Input follows the same structure as Binary Normalization



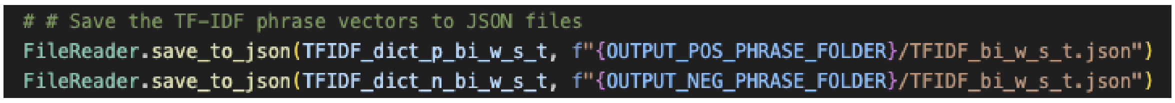
```python
# Input folders for n-gram and PoS + noun phrase reviews
INPUT_POS_PHRASE_FOLDER = "coursework/FeatureGeneration/Results/PhraseExtraction/pos"
INPUT_NEG_PHRASE_FOLDER = "coursework/FeatureGeneration/Results/PhraseExtraction/neg"
# Output folders for normalized n-grams and PoS + noun phrases
OUTPUT_POS_PHRASE_FOLDER = "coursework/FeatureGeneration/Results/Normalisation/phrases/pos"
OUTPUT_NEG_PHRASE_FOLDER = "coursework/FeatureGeneration/Results/Normalisation/phrases/neg"
```



```python
# Read bigrams, trigrams and noun phrases for tokenized words
pos_bigrams_white_space_tokenized = FileReader.load_json(f"{INPUT_POS_PHRASE_FOLDER}/bigrams_white_space_tokenized.json")
neg_bigrams_white_space_tokenized = FileReader.load_json(f"{INPUT_NEG_PHRASE_FOLDER}/bigrams_white_space_tokenized.json")
```



```python
# Compute TF-IDF using Scikit-Learn from bigrams, trigrams and noun phrases
TFIDF_dict_p_bi_w_s_t = compute_tfidf(pos_bigrams_white_space_tokenized, len(pos_bigrams_white_space_tokenized))
TFIDF_dict_n_bi_w_s_t = compute_tfidf(neg_bigrams_white_space_tokenized, len(neg_bigrams_white_space_tokenized))
```



```python
# # Save the TF-IDF phrase vectors to JSON files
FileReader.save_to_json(TFIDF_dict_p_bi_w_s_t, f"{OUTPUT_POS_PHRASE_FOLDER}/TFIDF_bi_w_s_t.json")
FileReader.save_to_json(TFIDF_dict_n_bi_w_s_t, f"{OUTPUT_NEG_PHRASE_FOLDER}/TFIDF_bi_w_s_t.json")
```

5.0 Feature Selection

This is the part of the project where we choose the best '*Feature Set*' on which we want to train on:

- This implementation has 5 main functions:
  - prepare_features(pos_data, neg_data):
  - train_naive_bayes(X, y, model_type):
  - evaluate_model(model, X_test, y_test):
  - load_test_reviews(test_folder):
  - main()
- **prepare_features(pos_data, neg_data):** We establish our training data *'X'* which combines our positive and negative feature dictionaries into a single dataset. To ensure we know which ones are positive and negative and assign labels to it denoted with *'y'*. This function also converts the feature dictionaries into a sparse matrix using *DictVectorizer()*.
- **train_naive_bayes(X, y, model_type):** Since we used 2 different types of Normalisation we have to use 2 different types of Naïve Bayes models. To train on Binary Normalization we use *BernoulliNB* while to train on TF-IDF Normalization we use *MultinomialNB*. This function hence outputs a trained Naïve Bayes Model.
- **evaluate_model(model, X_test, y_test):** This function helps predict the *accuracy* and the *F1* score for the given model. We use the pre-existing library *sklearn.metrics* to calculate these values.
- **load_test_reviews(test_folder):** As the name suggests this function reads the file from either the eval/test folder and since we append 'pos_' and 'neg_' to the respective positive and negative file names we use that to also create labels [1,0] so we can accurately calculate the accuracy, etc.
- **main():** This function essentially reads the 'training_data' from each normalised file we created and trains it by calling the other functions followed up by analysing the results. An 'if' statement was added to identify the different ways of how a file was normalized, and appended the accuracy and result to a separate output dictionary. At the end we save all the values into a .json file as a dictionary.

- **Results**

Boosted Token Feature Set	Accuracy %	F1
TFIDF_w_s_t	85.2	.852
BIN_ w_s_t	79.7	.795
TFIDF_l_t	82.0	.820
BIN_ l_t	80.2	.801
TFIDF_s_t	76.5	.765
BIN_ s_t	62.7	.586

Structure to read Feature Set Name:
[(Normalisation)_(phrase extraction)_(tokenisation)]

Keys for Token names:
bi(bigrams) | tri(trigrams) | np(noun phrases PoS tag) | w_s_t(white space tokenised) | s_t(stemmed tokenised) | l_t(lemmatized tokenised) | BIN(Binary Normalisation)

From these values we can clearly tell that TF-IDF performed far better than Binary Normalisation and so we shall go with using this, we can also note that tokenization through white space tokens did better than other tokenization and since boosted vs trigrams and bigrams result the same.
I chose TF-IDF with bigrams and using white space tokenisation to be my selected feature set

Phrase Feature Set	Accuracy %	F1
TFIDF_bi_w_s_t	85.2	.852
BIN_bi_w_s_t	74.3	.731
TFIDF_tri_w_s_t	85.2	.852
BIN_tri_w_s_t	74.3	.731
TFIDF_np_w_s_t	71.5	.711
BIN_np_w_s_t	69.3	.687
TFIDF_bi_l_t	82.8	.828
BIN_bi_l_t	75.2	.743
TFIDF_tri_l_t	82.8	.828
BIN_tri_l_t	74.7	.737
TFIDF_np_l_t	71.5	.715
BIN_np_l_t	70.2	.699
TFIDF_bi_s_t	76.3	.763
BIN_bi_s_t	56.0	.462
TFIDF_tri_s_t	76.5	.765
BIN_tri_s_t	55.3	.450
TFIDF_np_s_t	65.3	.647
BIN_np_s_t	64.7	.640

6.0 Naïve Bayes from Scratch

In this section I'll be showing my implementation of Naïve Bayes on the Selected Feature Set. The methodology will be specified per function below.

train_naive_bayes_from_scratch(pos_data, neg_data):
In this function we are doing 2 main tasks calculating priors (probability of positive/negative classes) and calculating likelihood (probability of terms per given class). Since I have split my training data into 2 folders 'pos' and 'neg' we are going to read each folder separately and represent the probability of each class (P(Positive) and P(Negative)).

$$P(Positive) = (Number\ of\ Positive\ Reviews)/\ (Total\ Number\ of\ Reviews)$$

$$P(Negative) = (Number\ of\ negative\ Reviews)/\ (Total\ Number\ of\ Reviews)$$

To calculate the Likelihoods of a term/word over both positive and negative reviews:

$$P(term|Positive) = (Count\ of\ term\ in\ Positive\ Reviews + \ Smoothing\ Factor/ \\ ((Total\ Number\ of\ terms\ in\ Positive\ Reviews + Smoothing\ Factor)\ \ X\ Vocabulary\ Size)$$

$$P(term|Negative) = (Count\ of\ term\ in\ negative\ Reviews + \ Smoothing\ Factor/ \\ ((Total\ Number\ of\ terms\ in\ Negative\ Reviews + Smoothing\ Factor)\ \ X\ Vocabulary\ Size)$$

The smoothing factor also called the *Laplace Smoothing Factor* ensures that no term has a 0 probability. In my implementation we use a set() to ensure all the tokens are unique, and we return 'prios_pos' and 'prior_neg' that returns the prior probability while 'likelihoods_pos' and 'likelihoods_neg' returns the likelihoods for both classes along with the Vocabulary.

predict_naive_bayes(review, prior_pos, prior_neg, likelihoods_pos, likelihoods_neg, vocab):
This function predicts a given review and checks if its either a positive or negative review. To calculate this, we can use Bayes Theorem:

$$P(\text{Class}|\text{Review}) \propto P(\text{Class}) \prod_{i=1}^{n} P(\text{term}_i|\text{Class})^{\text{count}_i}$$

Since we are multiplying multiple small probabilities, we must use the log of the probabilities.

$$\log P(\text{Class}|\text{Review}) \propto \log P(\text{Class}) + \sum_{i=1}^{n} \text{count}_i \cdot \log P(\text{term}_i|\text{Class})$$

At the end we get the probabilities of a review being either positive or negative; And if positive we return '1' else '0'.

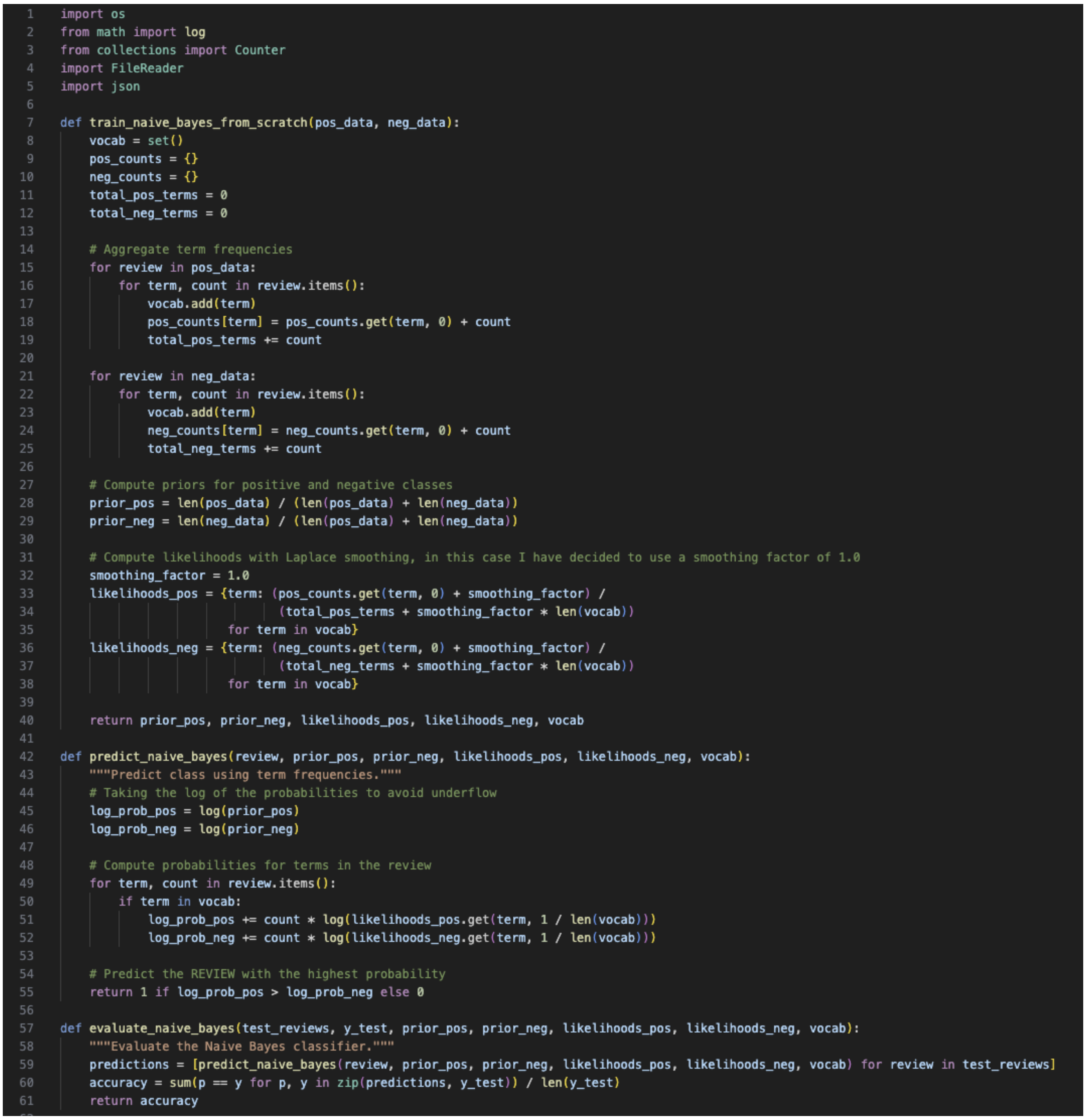
evaluate_naive_bayes(test_reviews, y_test, prior_pos, prior_neg, likelihoods_pos, likelihoods_neg, vocab):
Since in this task we are not allowed to use any external libraries we have to create a accuracy calculator from scratch which is essentially to check the probability is;
Number of correct predictions / Total number of predictions
We then compare this to the actual labels of each review and from there we can figure how many we got right and how many we got wrong.

loat_test_reviews():
Simply just loads all the reviews from a folder

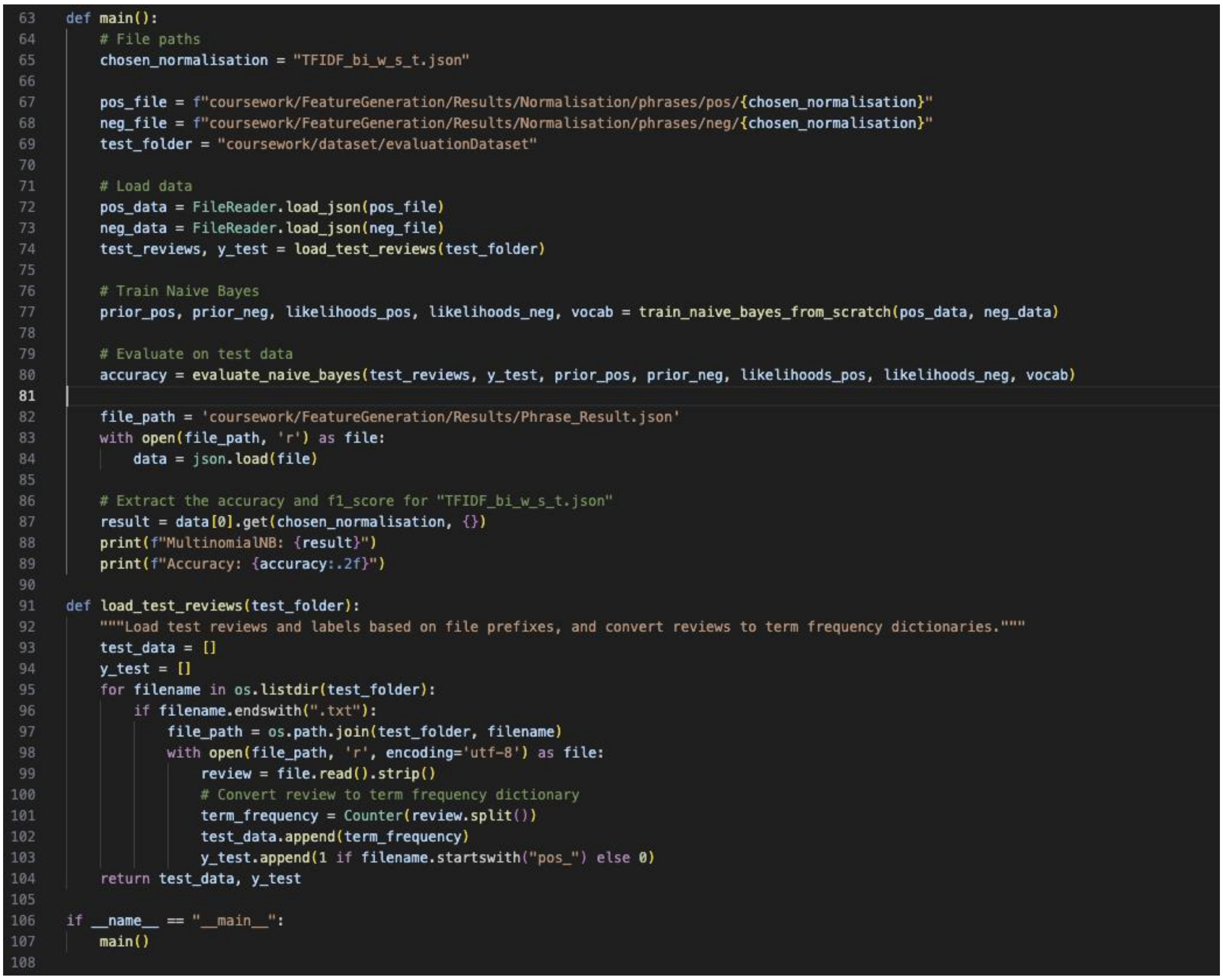
main():
Gives a clear structure when calling all the functions above and provides the end result

Method Used	Accuracy %
Using MultinomialNB	85.2
My implementation (can sometimes do better 😊)	83.0

Code:



```python
import os
from math import log
from collections import Counter
import FileReader
import json

def train_naive_bayes_from_scratch(pos_data, neg_data):
    vocab = set()
    pos_counts = {}
    neg_counts = {}
    total_pos_terms = 0
    total_neg_terms = 0

    # Aggregate term frequencies
    for review in pos_data:
        for term, count in review.items():
            vocab.add(term)
            pos_counts[term] = pos_counts.get(term, 0) + count
            total_pos_terms += count

    for review in neg_data:
        for term, count in review.items():
            vocab.add(term)
            neg_counts[term] = neg_counts.get(term, 0) + count
            total_neg_terms += count

    # Compute priors for positive and negative classes
    prior_pos = len(pos_data) / (len(pos_data) + len(neg_data))
    prior_neg = len(neg_data) / (len(pos_data) + len(neg_data))

    # Compute likelihoods with Laplace smoothing, in this case I have decided to use a smoothing factor of 1.0
    smoothing_factor = 1.0
    likelihoods_pos = {term: (pos_counts.get(term, 0) + smoothing_factor) /
                              (total_pos_terms + smoothing_factor * len(vocab))
                       for term in vocab}
    likelihoods_neg = {term: (neg_counts.get(term, 0) + smoothing_factor) /
                              (total_neg_terms + smoothing_factor * len(vocab))
                       for term in vocab}

    return prior_pos, prior_neg, likelihoods_pos, likelihoods_neg, vocab

def predict_naive_bayes(review, prior_pos, prior_neg, likelihoods_pos, likelihoods_neg, vocab):
    """Predict class using term frequencies."""
    # Taking the log of the probabilities to avoid underflow
    log_prob_pos = log(prior_pos)
    log_prob_neg = log(prior_neg)

    # Compute probabilities for terms in the review
    for term, count in review.items():
        if term in vocab:
            log_prob_pos += count * log(likelihoods_pos.get(term, 1 / len(vocab)))
            log_prob_neg += count * log(likelihoods_neg.get(term, 1 / len(vocab)))

    # Predict the REVIEW with the highest probability
    return 1 if log_prob_pos > log_prob_neg else 0

def evaluate_naive_bayes(test_reviews, y_test, prior_pos, prior_neg, likelihoods_pos, likelihoods_neg, vocab):
    """Evaluate the Naive Bayes classifier."""
    predictions = [predict_naive_bayes(review, prior_pos, prior_neg, likelihoods_pos, likelihoods_neg, vocab) for review in test_reviews]
    accuracy = sum(p == y for p, y in zip(predictions, y_test)) / len(y_test)
    return accuracy
```



```python
def main():
    # File paths
    chosen_normalisation = "TFIDF_bi_w_s_t.json"

    pos_file = f"coursework/FeatureGeneration/Results/Normalisation/phrases/pos/{chosen_normalisation}"
    neg_file = f"coursework/FeatureGeneration/Results/Normalisation/phrases/neg/{chosen_normalisation}"
    test_folder = "coursework/dataset/evaluationDataset"

    # Load data
    pos_data = FileReader.load_json(pos_file)
    neg_data = FileReader.load_json(neg_file)
    test_reviews, y_test = load_test_reviews(test_folder)

    # Train Naive Bayes
    prior_pos, prior_neg, likelihoods_pos, likelihoods_neg, vocab = train_naive_bayes_from_scratch(pos_data, neg_data)

    # Evaluate on test data
    accuracy = evaluate_naive_bayes(test_reviews, y_test, prior_pos, prior_neg, likelihoods_pos, likelihoods_neg, vocab)

    file_path = 'coursework/FeatureGeneration/Results/Phrase_Result.json'
    with open(file_path, 'r') as file:
        data = json.load(file)

    # Extract the accuracy and f1_score for "TFIDF_bi_w_s_t.json"
    result = data[0].get(chosen_normalisation, {})
    print(f"MultinomialNB: {result}")
    print(f"Accuracy: {accuracy:.2f}")

def load_test_reviews(test_folder):
    """Load test reviews and labels based on file prefixes, and convert reviews to term frequency dictionaries."""
    test_data = []
    y_test = []
    for filename in os.listdir(test_folder):
        if filename.endswith(".txt"):
            file_path = os.path.join(test_folder, filename)
            with open(file_path, 'r', encoding='utf-8') as file:
                review = file.read().strip()
                # Convert review to term frequency dictionary
                term_frequency = Counter(review.split())
                test_data.append(term_frequency)
                y_test.append(1 if filename.startswith("pos_") else 0)
    return test_data, y_test

if __name__ == "__main__":
    main()
```

7.0 BERT

In this section we look at how we can implement BERT to predict positive and negative reviews.
In simple words Cased Vs Uncased models simply revolves around how to handle case sensitivity in text. Going into my testing my hypothesis was that Uncased would fare better than Cased. Since we are specifically looking at only the terms and not its case there would be a higher chance that tokens are matched compared to Cased. This however led to an interesting result.

BERT Model Type	Epoch 1	Epoch 2	Epoch 3
Cased	88	89	91
Uncased	90.83	89.2	90.9

Unfortunately, due to having low computational power, I was only able to test it on 3 Epochs. However we can note that training progressively increased on cased while the training loss remained relatively constant for Uncased BERT.