IST 664 Natural Language Processing
Project Report

MOVIE REVIEW SENTIMENT CLASSIFICATION

Team members:
1. Resham Bahira
2. Nitihin Kumar Hadhge
Girish Kumar

1	**Introduction**
2.	**Data Collection**
3.	**Data preprocessing**
4.	**Feature engineering**
5.	**Saving Feature Sets to CSV**
6.	**Data Visualization**
7.	**Experiments**
8.	**Conclusion**

1.Introduction

DATASET

The dataset, created for the Kaggle competition, combines data from Socher et al.'s sentiment analysis and the original movie review corpus by Pang and Lee, sourced from Rotten Tomatoes. Socher's team employed crowd-sourcing to manually annotate sub-sentences with sentiment labels, including "Negative", "Little Negative", "Neutral", "Little Positive", and "Positive". The dataset is divided into training and testing sets, with sentiment-labeled sentences in the train.tsv file and unlabeled sentences in the test.tsv file. Each sentence is associated with one of the following sentiment labels:

- 0 - Negative
- 1 - Little Negative
- 2 - Neutral
- 3 - Little Positive
- 4 - Positive

2.READING DATA FROM .CSV FILE

The main function takes two command-line arguments when executed. The first argument specifies the directory path where the train and test files are located, and the second argument defines the sample size. Within this function, the `processkaggle` function is called with these arguments. The `processkaggle` function handles initial tasks such as splitting the file into individual lines, then proceeds to call the preprocessing and feature set functions.

```
# function to read kaggle training file, train and test a classifier
# function to read kaggle training file, train and test a classifier
# function to read kaggle training file, train and test a classifier
def processkaggle(dirPath,limitStr):
  # convert the limit argument from a string to an int
  limit = int(limitStr)

  os.chdir(dirPath)

  f = open('/Users/nithinkumar/Desktop/FinalProjectData/corpus/train.tsv', 'r')
  # loop over lines in the file and use the first limit of them
  phrasedata = []
  for line in f:

    # ignore the first line starting with Phrase and read all lines
    if (not line.startswith('Phrase')):
      # remove final end of line character
      line = line.strip()
      # each line has 4 items separated by tabs
      # ignore th
      # e phrase and sentence ids, and keep the phrase and sentiment
      phrasedata.append(line.split('\t')[2:4])

Run | Debug
if __name__ == '__main__':
    if (len(sys.argv) != 3):
        print ('usage: classifyKaggle.py <corpus-dir> <limit>')
        sys.exit(0)
    processkaggle(sys.argv[1], sys.argv[2])
```

3.PREPROCESSING AND FILTERING DATA

Both processed and unprocessed data were used in all the experiments conducted.

Removing Punctuation

```
        cleaned_token = re.sub(r'[^\w\s]', '', token)
```

This step uses regular expressions to remove punctuation from the tokens. The re.sub(r'[^\w\s]', '', token) function removes any characters that are not words (\w) or whitespace (\s). This means characters like commas, periods, and other punctuation marks are removed from the tokenized words.

Punctuation marks generally do not carry significant meaning in sentiment analysis, so removing them helps reduce noise and ensures the model focuses on the meaningful content in the text.

Removing Stop Words

```
if cleaned_token and cleaned_token not in stopwords_set:
```

A stopwords set is used to filter out common words like "the", "is", "in", etc., that do not provide meaningful insight into sentiment. The stopwords set is expanded beyond the default NLTK stopwords list by including custom words that are deemed unnecessary for analysis.

Stopwords are removed because they don't contribute much to the overall meaning or sentiment of the text. By eliminating these words, the model can focus on more informative terms.

Filtering Short Tokens

```
if len(lemmatized_token) > 2:
```

This condition filters out tokens that are shorter than two characters. Words like "em", "nt", and other informal abbreviations or irrelevant terms are excluded. These short tokens are typically noise in the text that don't provide useful information.

Removing short tokens helps clean the data further by removing unwanted abbreviations or incomplete words that don't contribute to the analysis.

Lemmatization

```
lemmatized_token = lemmatizer.lemmatize(cleaned_token)
```

Lemmatization is a technique that reduces words to their base or root form. For example, "running" becomes "run", and "better" becomes "good". This helps consolidate different forms of a word into a single representative form.

Lemmatization ensures that similar words are treated the same, reducing complexity and improving the model's ability to detect patterns or sentiments across different variations of a word.

Adding Bigrams to Final Tokens

```
# Add bi-grams
bi_grams = list(ngrams(cleaned_tokens, 2))
bi_grams = ['_'.join(bigram) for bigram in bi_grams]
final_tokens = cleaned_tokens + bi_grams
```

After cleaning the tokens (removing punctuation, stopwords, and short words), bigrams (pairs of adjacent words) are generated. These bigrams are combined into strings like "good_morning". The final_tokens list combines both the cleaned tokens and the generated bigrams.

Including bigrams (and other n-grams) helps capture word combinations or phrases that carry specific sentiment or meaning, which individual words alone may miss.

Filtering Tokens (Final Step)

```
def filtered_tokens(phrase):
    tokens, sentiment = phrase  # Assuming `phrase` is a tuple of (tokens, sentiment)
    filtered = [token for token in tokens if token not in stopwords_set]
    return (filtered, sentiment)
```

This function takes a tuple (tokens, sentiment) and filters out stopwords from the list of tokens. The function returns the cleaned list of tokens along with the original sentiment label, assuming that sentiment is associated with the tokens.

This final filtering step ensures that tokens which are not meaningful (stopwords) are removed while retaining the sentiment of the original data. This is critical for machine learning tasks where the relationship between features (tokens) and labels (sentiment) needs to be preserved.

Extending the Stopwords List

```
# Comprehensive stopwords list
def get_extended_stopwords():
    nltkstopwords = set(stopwords.words('english'))
    custom_stopwords = {'could', 'would', 'might', 'must', 'need', 'sha', 'wo', 'y', "'s", "'d",
                        "'ll", "'t", "'m", "'re", "'ve", "n't", "'i", 'not', 'no', 'can', 'don',
                        'nt', 'actually', 'also', 'always', 'even', 'ever', 'just', 'really',
                        'still', 'yet', 'however', 'nevertheless', 'furthermore', 'therefore',
                        'otherwise', 'meanwhile', 'though', 'although', 'thus', 'hence', 'indeed',
                        'perhaps', 'especially', 'specifically', 'usually', 'often', 'sometimes',
                        'certainly', 'typically', 'mostly', 'generally', 'about', 'above', 'across',
                        'after', 'against', 'among', 'around', 'at', 'before', 'behind', 'below',
                        'beneath', 'beside', 'between', 'beyond', 'during', 'inside', 'onto',
                        'outside', 'through', 'under', 'upon', 'within', 'without'}
    return nltkstopwords.union(custom_stopwords)
```

This function extends the list of stopwords by adding custom words to the NLTK's default stopwords list. These custom words are terms that may not add significant meaning to the analysis, so they are excluded.

Extending the stopwords list helps ensure that additional unnecessary words, which may be context-specific or domain-specific, are removed from the analysis, further improving the model's performance.

At the end of the preprocessing pipeline, all the steps—lowercase conversion, tokenization, punctuation removal, stopword filtering, short token filtering, lemmatization, and bigram generation—are encapsulated in a single function, preprocessing(). This function takes a line of text, applies all the cleaning and processing steps, and returns the final set of tokens along with bigrams, ready for further analysis or model training. This approach ensures that the entire preprocessing process is streamlined and handled in one place, making it easy to apply to the dataset consistently.

```
# Enhance preprocessing with bi-grams and sentiment scores
def preprocessing(line):
    # Tokenize the input line after converting it to lowercase.
    tokens = word_tokenize(line.lower())
    cleaned_tokens = []
    # Process each token to remove punctuation, check against stopwords, and lemmatize.
    for token in tokens:
        # Remove punctuation from the token.
        cleaned_token = re.sub(r'[^\w\s]', '', token)

        # Check if the cleaned token is not in the stopwords list.
        if cleaned_token and cleaned_token not in stopwords_set:
            # Lemmatize the token to reduce it to its base form.
            lemmatized_token = lemmatizer.lemmatize(cleaned_token)

            # Only include tokens longer than 2 characters.
            if len(lemmatized_token) > 2:
                cleaned_tokens.append(lemmatized_token)
    # Add bi-grams
    # Generate bi-grams from the list of cleaned and lemmatized tokens.
    bi_grams = list(ngrams(cleaned_tokens, 2))
    # Join the words in each bi-gram with an underscore.
    bi_grams = ['_'.join(bigram) for bigram in bi_grams]
    # Combine the single tokens and bi-grams into one list.
    final_tokens = cleaned_tokens + bi_grams
    # Return all tokens as a single string separated by spaces.
    return ' '.join(final_tokens)

def filtered_tokens(phrase):
    # Extract tokens and sentiment score from the input tuple.
    tokens, sentiment = phrase # Assuming `phrase` is a tuple of (tokens, sentiment)
    # Filter out tokens that are in the stopwords set.
    filtered = [token for token in tokens if token not in stopwords_set]
    # Return the filtered tokens and the original sentiment as a tuple.
    return (filtered, sentiment)
```

4. GENERATING FEATURE SETS:

Different functions have been used to generate feature sets for both pre-processed and unprocessed data. These functions help create two lists: one for pre-processed tokens and one for unprocessed tokens. These lists are then used for generating filtered lists for pre-processed tokens and lists for unprocessed tokens.

```
## Initialize lists to hold processed and unprocessed data.
withpreprocessing = []
withoutpreprocessing= []
# Iterate through each phrase and sentiment tuple in the phraselist.
for p in phraselist:
  # Tokenize the first element (phrase) of the tuple using NLTK's word_tokenize
  tokens = nltk.word_tokenize(p[0])
  # Append the tokenized phrase and its associated integer sentiment score to the list without preprocessing.
  withoutpreprocessing.append((tokens, int(p[1])))

  # Apply the preprocessing function to the phrase and re-tokenize the result.
  p[0] = preprocessing(p[0]) # Preprocess the text of the phrase.
  tokens = nltk.word_tokenize(p[0])
  # Append the re-tokenized, preprocessed phrase and its sentiment to the list with preprocessing.
  withpreprocessing.append((tokens, int(p[1])))

# Initialize an empty list to store preprocessed data after further filtering.
withpreprocessing_filter=[]
# Copy each element from the list 'withpreprocessing' to 'withpreprocessing_filter'.
for p in withpreprocessing:
  withpreprocessing_filter.append(p)
# Initialize lists to store all tokens collected from processed data.
filtered_tokens =[]
unfiltered_tokens = []
# Iterate over the list that contains preprocessed data.
for (d,s) in  withpreprocessing_filter:
  # Collect all tokens from the tuple where 'd' is a list of tokens and 's' is the sentiment score.
  for i in d:
    filtered_tokens.append(i)
# Iterate over the list containing unprocessed data.
for (d,s) in withoutpreprocessing:
  # Collect all tokens from the tuple where 'd' is a list of tokens and 's' is the sentiment score.
  for i in d:
    unfiltered_tokens.append(i)
```

Bag of Words:

This method involves representing each document as a set of unique words (tokens) without considering the order. The Bag of Words model helps capture the presence of words in the document, making it easier to analyze the frequency of words and their contribution to the sentiment.

```
# Different Functions for feature sets :
#Bag of wordds
def bw(a,i):
  # Calculate the frequency distribution of elements in the list 'a'.
  a = nltk.FreqDist(a)
  # Extract the 'i' most common elements from the frequency distribution.
  # 'w' represents the word and 'c' represents its count in the list.
  wf = [w for (w,c) in a.most_common(i)]
  # Return the list of the most frequent words
  return wf

# Apply the 'bw' function to 'filtered_tokens' to obtain the 350 most frequent words.
filtered_bow_features = bw(filtered_tokens,350)
# Apply the 'bw' function to 'unfiltered_tokens' to obtain the 350 most frequent words.
unfiltered_bow_features = bw(unfiltered_tokens,350)
```

Unigram:

The functions bw(a, i) and uf(d, wf) work together to extract unigram features from text data.

1. **bw(a, i)**:
    - This function takes a list of words a and returns the i most frequent words from that list. It uses NLTK's FreqDist to compute the frequency distribution of words and then selects the top i most common words.
2. **uf(d, wf)**:
    - This function creates a binary feature set for a document d based on the most frequent words wf identified by the bw function.
    - It checks whether each word in wf appears in d and returns a dictionary where the keys are the words and the values are 1 if the word is present in the document, and 0 if it is absent.

Together, these functions extract unigram features by first identifying the most frequent words and then checking their presence in a document, creating a feature set for further analysis or modelling.

```python
#Unigram
def uf(d,wf):
  # Create a set from the list 'd' to allow faster membership testing.
  df= set(d)
  # Initialize an empty dictionary to store unigram features.
  f = {}
  # Loop through each word in the list of words 'wf'
  for word in wf:
    # Create a feature key for each word and assign a boolean value
    # indicating whether the word is present in the data 'd'.
    f['V_%s'% word] = (word in df)
  # Return the dictionary containing feature presence information.
  return f

# Generate unigram features for each tuple in 'withpreprocessing_filter' using the 'filtered_tokens' as the vocabulary.
# 'd' is the list of tokens, and 's' is the sentiment score from each tuple in 'withpreprocessing_filter'.
filtered_unigram_features = [(uf(d,filtered_tokens),s) for (d,s) in withpreprocessing_filter]
# Similarly, generate unigram features for each tuple in 'withoutpreprocessing' using the 'unfiltered_tokens' as the vocabulary.
# This considers all original tokens without preprocessing to compare the effect.
unfiltered_unigram_features = [(uf(d,unfiltered_tokens),s) for (d,s) in withoutpreprocessing]
```

Bigram:

The **bigram_bow(wordlist, n)** function is used to extract the top n most significant bigrams from a list of words, based on their frequency and statistical association. It first initializes the BigramAssocMeasures to assess the strength of bigrams. Then, using NLTK's BigramCollocationFinder, it generates bigrams from the provided wordlist and applies a frequency filter to only include bigrams that occur at least twice. The function then ranks the bigrams based on the chi-square statistic and returns the top n most significant bigrams as the output, which are useful for capturing word pairs that frequently appear together in the text.

The **bf(doc, word_features, bigram_feature)** function creates a binary feature set for a document doc by checking the presence of specified word and bigram features. It first converts the document into a set dw for efficient look-up and generates bigrams from the document using NLTK's bigrams function. Then, for each word in word_features, it checks if the word is present in dw and assigns a value of 1 if it is, otherwise 0. Similarly, for each bigram in bigram_feature, it checks if the bigram exists in the document's bigrams db and creates a feature indicating its presence or absence. This binary feature representation is useful for machine learning models, where the presence of specific words or bigrams can be used to classify or analyze the document.

These functions are applied to both filtered and unfiltered data.

```python
def bigram_bow(wordlist,n):
  # Utilize NLTK to measure bigram associations.
  bigram_measure = nltk.collocations.BigramAssocMeasures()
  # Create a BigramCollocationFinder from a list of words
  finder = BigramCollocationFinder.from_words(wordlist)
  # Apply a frequency filter to bigrams that appear at least 2 times.
  finder.apply_freq_filter(2)
  # Find the best 4000 bigrams using the chi-squared association measure.
  b_features = finder.nbest(bigram_measure.chi_sq,4000)
  # Return the top 'n' bigrams from the filtered list.
  return b_features[:n]


def bf(doc,word_features,bigram_feature):
  # Create a set of words and a set of bigrams from the document for fast lookup.
  dw = set(doc)
  db = nltk.bigrams(doc)
  # Initialize an empty dictionary to store features
  features = {}
  # Check each word in the word_features list and add it as a binary feature
  for word in word_features:
    features['V_{}'.format(word)] = (word in dw)
  # Check each bigram in the bigram_feature list and add it as a binary feature
  for b in bigram_feature:
    features['B_{}_{}'.format(b[0],b[1])] = (b in db)
  # Return the dictionary containing all word and bigram features.
  return features

# Generate bigram features for each tuple in 'withpreprocessing_filter' using filtered tokens and bow features.
filtered_bigram_features = [(bf(d,filtered_bow_features,bigram_bow(filtered_tokens,350)),s) for (d,s) in withpreprocessing_filter]
# Similarly, generate bigram features for each tuple in 'withoutpreprocessing' using unfiltered tokens and bow features.
unfiltered_bigram_features = [(bf(d,unfiltered_bow_features,bigram_bow(unfiltered_tokens,350)),s) for (d,s) in withoutpreprocessing]
```

Bigrams (pairs of words) provide context and improve understanding of sentiment by capturing relationships between adjacent words, which single words (unigrams) alone cannot convey.

POS Tagging:

The pf(document, word_features) function generates a feature set for a given document by checking the presence of specific words and analyzing the parts of speech (POS) distribution within the document.

1. Word Presence Check:The function first converts the document into a set of unique words (document_words). Then, for each word in the word_features list, it checks if that word exists in the document. If a word is present, it adds a feature with a value of 1, otherwise 0. This helps in identifying the presence or absence of certain important words in the document.
2. POS Tagging and Count:The function then uses NLTK's pos_tag to tag each word in the document with its respective part of speech (POS). It initializes counters for nouns, verbs, adjectives, and adverbs. As it iterates through the tagged words, it increments the respective counters based on the POS tag:
    - Nouns (tag starts with 'N')
    - Verbs (tag starts with 'V')
    - Adjectives (tag starts with 'J')

        - Adverbs (tag starts with 'R')
    - Finally, it adds the counts of these POS categories as features in the feature set.

The output is a dictionary with features that indicate whether specific words from word_features are present in the document and the count of nouns, verbs, adjectives, and adverbs in the document. This feature set can be used in various NLP tasks, such as sentiment analysis or text classification.

```
#POs Tags
def pf(document, word_features):
    # Create a set of all words in the document for fast lookup.
    document_words = set(document)
    # Generate POS tags for each word in the document.
    tagged_words = nltk.pos_tag(document)
    # Initialize an empty dictionary to store features.
    features = {}
    # Loop through each word in word_features and add it as a binary feature indicating presence in the document.
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    # Initialize counters for different POS tags.
    numNoun = 0
    numVerb = 0
    numAdj = 0
    numAdverb = 0
    # Count occurrences of each major part of speech.
    for (word, tag) in tagged_words:
        if tag.startswith('N'): numNoun += 1
        if tag.startswith('V'): numVerb += 1
        if tag.startswith('J'): numAdj += 1
        if tag.startswith('R'): numAdverb += 1
    # Store counts of each part of speech in the features dictionary.
    features['nouns'] = numNoun
    features['verbs'] = numVerb
    features['adjectives'] = numAdj
    features['adverbs'] = numAdverb
    # Return the dictionary containing all the features.
    return features

# Generate POS tag features for each tuple in 'withpreprocessing_filter' using the filtered BOW features.
filtered_pos_features = [(pf(d,filtered_bow_features),s) for (d,s) in withpreprocessing_filter]
# Similarly, generate POS tag features for each tuple in 'withoutpreprocessing' using the unfiltered BOW features.
unfiltered_pos_features = [(pf(d,unfiltered_bow_features),s) for (d,s) in withoutpreprocessing]
```

Sentiment Lexicon:

The **slf(document, word_features, SL)** function generates a feature set for a document based on the presence of specific words and sentiment analysis using a sentiment lexicon (SL).

1. **Word Presence**: It checks if each word from the word_features list appears in the document, adding binary features (1 for presence, 0 for absence) for each word.

2. **Sentiment Analysis**: For each word in the document, it looks up the word in the sentiment lexicon (SL) to retrieve its sentiment strength, part of speech, and polarity. It then counts the occurrences of different sentiment types (positive, negative, neutral) and stores these counts as features.
3. **Output**: The function returns a feature dictionary that includes the binary features for the specified words and the counts of positive, negative, and neutral sentiments in the document. This feature set is useful for sentiment-based tasks such as classification or analysis.

```
#Sentiment Lexicon Features
def slf(document, word_features, SL):
    # Create a set of all words in the document for quick lookup.
    document_words = set(document)
    # Create dictionary features with keys like 'V_word' where each key indicates whether a word is present in the document.
    features = {f'V_{word}': (word in document_words) for word in word_features}
    # Initialize a dictionary to count occurrences of sentiment-related terms.
    sentiment_counts = {}
    # Loop through each word in the document.
    for word in document_words:
        # Check if the word is in the sentiment lexicon (SL).
        if word in SL:
            # Extract sentiment attributes for the word.
            strength, posTag, isStemmed, polarity = SL[word]
            # Create a composite key from the strength and polarity (e.g., 'strongPositive').
            key = f'{strength}{polarity.capitalize()}'
            # Initialize the count for this key if it's not already present.
            if key not in sentiment_counts:
                sentiment_counts[key] = 0
            # Increment the count for this sentiment key.
            sentiment_counts[key] += 1

    # Identify keys that indicate positive, negative, and neutral sentiments.
    positive_keys = [key for key in sentiment_counts if 'Positive' in key]
    negative_keys = [key for key in sentiment_counts if 'Negative' in key]
    neutral_keys = [key for key in sentiment_counts if 'Neutral' in key]

    # Calculate the total counts for positive, negative, and neutral terms.
    features['positivecount'] = sum(sentiment_counts[key] for key in positive_keys)
    features['negativecount'] = sum(sentiment_counts[key] for key in negative_keys)
    features['neutralcount'] = sum(sentiment_counts[key] for key in neutral_keys if 'Neutral' in key)
    # Return the dictionary containing all the features.
    return features
# Generate sentiment-related features for each tuple in 'withpreprocessing_filter' using the filtered BOW features and a sentiment lexicon (SL).
filtered_sl_features = [(slf(d, filtered_bow_features, SL), c) for (d, c) in withpreprocessing_filter]
# Similarly, generate sentiment-related features for each tuple in 'withoutpreprocessing' using the unfiltered BOW features and the same sentiment lexicon (SL).
unfiltered_sl_features = [(slf(d, unfiltered_bow_features, SL), c) for (d, c) in withoutpreprocessing]
```

LIWC Feature set:

The **liwc(doc, word_features, poslist, neglist)** function helps in sentiment analysis by providing valuable features based on word presence and sentiment categorization. By checking if words from a predefined list of relevant terms (word_features) appear in the document, it creates binary features that indicate the presence or absence of these key terms, which can help in determining the document's overall content and context. Additionally, by counting the occurrences of positive and negative words from the LIWC sentiment lists, it helps capture the emotional tone of the document. This sentiment data can be particularly useful for analyzing the document's sentiment, such as whether it leans positive, negative, or neutral, and aids in building predictive models for sentiment classification. The inclusion of LIWC-based sentiment features enhances the model's ability to detect nuanced emotional expressions and contributes to more accurate sentiment analysis.

This function is applied to both filtered and unfiltered data to extract features for sentiment classification. Extracting LIWC features from both filtered and unfiltered tokens ensures that sentiment-related emotional information is captured from all versions of the data, improving model accuracy.

```
#Linguistic Inquiry and Word Count
def liwc(doc,word_features,poslist,neglist):
  # Create a set of words from the document to eliminate duplicates and allow for fast lookup.
  doc_words = set(doc)
  # Initialize a dictionary to store features.
  features= {}
  # Loop through each word in word_features and add it as a binary feature indicating presence in the document.
  for word in word_features:
    features['contains({})'.format(word)] = (word in doc_words)
  # Initialize counters for positive and negative words.
  pos = 0
  neg = 0
  # Count occurrences of words listed in the positive and negative lists.
  for word in doc_words:
    if sentiment_read_LIWC_pos_neg_words.isPresent(word,poslist):
      pos+=1
    elif sentiment_read_LIWC_pos_neg_words.isPresent(word,neglist):
      neg+=1
    # Store the counts of positive and negative words in the features dictionary.
    features ['positivecount'] = pos
    features ['negativecount'] = neg

  # Ensure that features for positive and negative counts are present, even if their count is zero.
  if 'positivecount' not in features:
    features['positivecount'] = 0
  if 'negativecount' not in features:
    features['negativecount'] = 0
  # Return the dictionary containing all the features.
  return features

# Apply the liwc function to each document in the preprocessed  data sets.
filtered_liwc_features = [(liwc(d, filtered_bow_features, poslist,neglist), c) for (d, c) in withpreprocessing_filter]
# Apply the liwc function to each document in the unprocessed data sets.
unfiltered_liwc_features = [(liwc(d, unfiltered_bow_features, poslist,neglist), c) for (d, c) in withoutpreprocessing]
```

Combination of LIWC and SL:

The combined feature extraction method merges both LIWC (Linguistic Inquiry and Word Count) and SL (subjectivity lexicon) features. In this approach, strong positive and strong negative features are counted twice, as they are identified by both LIWC and SL. However, weak positive and weak negative features are only counted through the SL method. Combining LIWC and SL features takes advantage of both methods' strengths, providing a more comprehensive understanding of sentiment. Counting strong sentiment features twice ensures their importance is reinforced, while weak sentiment features are still captured but without redundancy.

For both filtered and unfiltered tokens, features were generated. Generating features for both types of data allows the model to compare the impact of preprocessing on sentiment analysis and determine the best approach for classification.

```python
def combo(doc, word_features, SL, poslist, neglist):
    # Create a set of words from the document to eliminate duplicates and allow for fast lookup.
    doc_words = set(doc)
    # Generate a dictionary of binary features indicating the presence of specific words from word_features.
    features = {f'contains({word})': (word in doc_words) for word in word_features}
    # Initialize a dictionary to keep track of different sentiment counts.
    sentiment_counts = {}
    # Count the occurrences of words from poslist and neglist as strong positive or negative sentiments.
    for word in doc_words:
        if word in poslist:
            sentiment_counts['strongPos'] = sentiment_counts.get('strongPos', 0) + 1
        elif word in neglist:
            sentiment_counts['strongNeg'] = sentiment_counts.get('strongNeg', 0) + 1
        # If the word is in the sentiment lexicon, analyze its sentiment attributes.
        if word in SL:
            strength, posTag, isStemmed, polarity = SL[word]
            key = f'{strength}{polarity.capitalize()}'
            if key not in sentiment_counts:
                sentiment_counts[key] = 0
            sentiment_counts[key] += 1

    # Gather keys that indicate positive, negative, and neutral sentiments.
    positive_keys = [key for key in sentiment_counts if 'Positive' in key]
    negative_keys = [key for key in sentiment_counts if 'Negative' in key]
    neutral_keys = [key for key in sentiment_counts if 'Neutral' in key]

    # Calculate the total counts for positive, negative, and neutral terms.
    features['positivecount'] = sum(sentiment_counts[key] for key in positive_keys)
    features['negativecount'] = sum(sentiment_counts[key] for key in negative_keys)
    features['neutralcount'] = sum(sentiment_counts[key] for key in neutral_keys if 'Neutral' in key)
    # Return the dictionary containing all the features.
    return features

# Apply the combo function to each document in the preprocessed datasets.
filtered_combo_features =  [(combo(d, filtered_bow_features,SL, poslist,neglist), c) for (d, c) in withpreprocessing_filter]
# Apply the combo function to each document in the unprocessed datasets.
unfiltered_combo_features = [(combo(d, unfiltered_bow_features,SL, poslist,neglist), c) for (d, c) in withoutpreprocessing]
```

Saving Feature Sets to CSV Files:

All generated feature sets have been saved into CSV files for future use as training sets with other classifiers or in separate Python notebooks. This ensures the feature sets are readily available for modelling, even if computational constraints prevent their immediate use in another Python script.

Saving feature sets to CSV files helps avoid recomputing the features each time, saving time and computational resources. It also makes the data easily accessible for future experiments or model training.

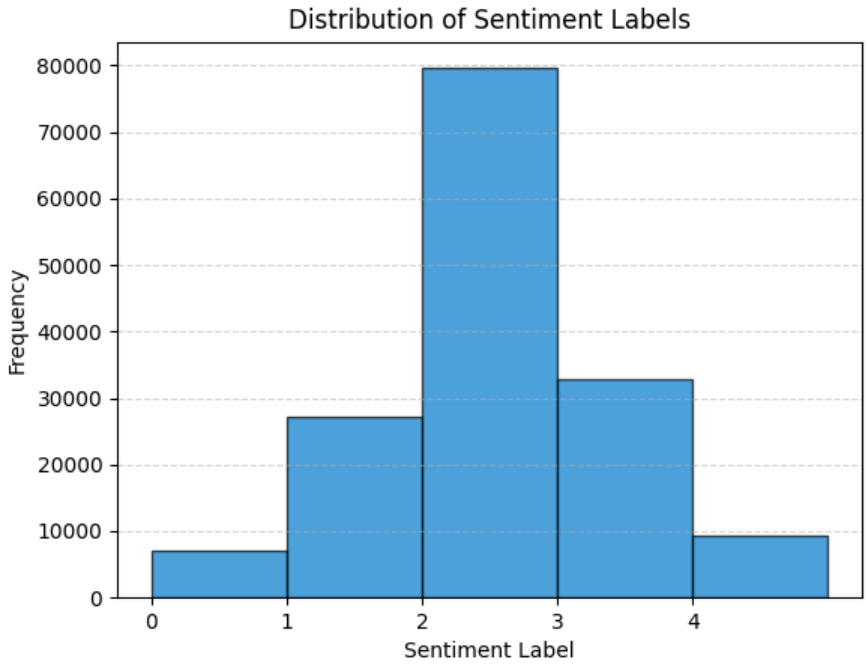
```
# Saving feature sets for for other classifier training
def save(features, path):
    # Open a file for writing at the specified path.
    f = open(path, 'w')
    # Retrieve the names of the features from the first feature set in the list.
    featurenames = features[0][0].keys()
    # Prepare the header line by cleaning up feature names and appending them into a single string.
    fnameline = ''
    for fname in featurenames:
        # Replace problematic characters in feature names to avoid issues in CSV format.
        fname = fname.replace(',','COM')
        fname = fname.replace("'","SQ")
        fname = fname.replace('"','DQ')
        fnameline += fname + ','
    # Add a label for the class/category column at the end of the header.
    fnameline += 'Level'
    f.write(fnameline)
    f.write('\n')
    # Iterate over each feature set and corresponding class/category.
    for fset in features:
        featureline = ''
        # Gather all feature values into a line, handling missing keys if necessary.
        for key in featurenames:
            # Check if the key exists in the feature set
            if key in fset[0]:
                featureline += str(fset[0][key]) + ','
            else:
                featureline += 'NA,'  # Write 'NA' for missing features in some records.
        # Append the class/category description based on the numerical label.
        if fset[1] == 0:
          featureline += str("Less Negitive")
        elif fset[1] == 1:
          featureline += str("Strong negitive")
        elif fset[1] == 2:
          featureline += str("Neutral")
        elif fset[1] == 3:
          featureline += str("Strongly positive")
        elif fset[1] == 4:
          featureline += str("Less positive")
        # Write the complete line for the current feature set to the file
        f.write(featureline)
        f.write('\n')
    # Close the file after writing all the data.
    f.close()
```

```
## Save unigram features from filtered data to 'filtered_unigram.csv'.
save(filtered_unigram_features,'filtered_unigram.csv')
# Save unigram features from unfiltered data to 'unfiltered_unigram.csv'.
save(unfiltered_unigram_features,'unfiltered_unigram.csv')

# Save bigram features from filtered data to 'filtered_bigram.csv'.
save(filtered_bigram_features,'filtered_bigram.csv')
# Save bigram features from unfiltered data to 'unfiltered_bigram.csv'.
save(unfiltered_bigram_features,'unfiltered_bigram.csv')

# Save POS tag-based features from filtered data to 'filtered_pos.csv'.
save(filtered_pos_features,'filtered_pos.csv')
#Save POS tag-based features from unfiltered data to 'unfiltered_pos.csv'.
save(unfiltered_pos_features,'unfiltered_pos.csv')

# Save sentiment lexicon features from filtered data to 'filtered_sl.csv'.
save(filtered_sl_features,'filtered_sl.csv')
# Save sentiment lexicon features from unfiltered data to 'unfiltered_sl.csv'.
save(unfiltered_sl_features,'unfiltered_sl.csv')

# Save LIWC-based features from filtered data to 'filtered_liwc.csv'.
save(filtered_liwc_features,'filtered_liwc.csv')
# Save LIWC-based features from unfiltered data to 'unfiltered_liwc.csv'.
save(unfiltered_liwc_features,'unfiltered_liwc.csv')

# Save combined sentiment and word presence features from filtered data to 'filtered_combo.csv'.
save(filtered_combo_features,'filtered_combo.csv')
# Save combined sentiment and word presence features from unfiltered data to 'unfiltered_combo.csv'.
save(unfiltered_combo_features,'unfiltered_combo.csv')
```

5.DATA VISUALIZATION:

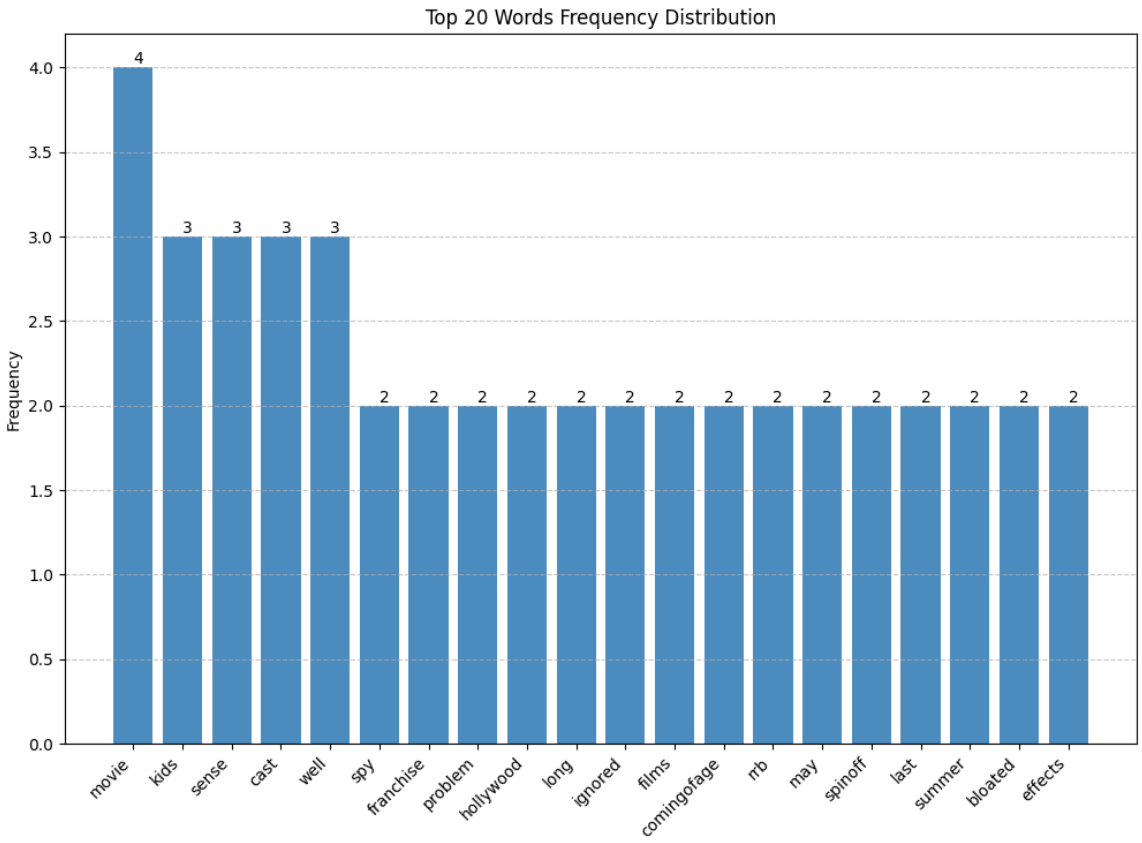
Sentiment Distribution Histogram

This histogram visualizes the distribution of sentiment labels in the dataset, showing how frequently each label occurs. The height of each bar represents the count of occurrences for a particular sentiment label. It reveals that neutral sentiment (label 2) has the highest frequency, followed by "little positive" (label 3) and "little negative" (label 1). Extreme sentiments, "negative" (label 0) and "positive" (label 4), occur less frequently, highlighting the dataset's overall sentiment composition.



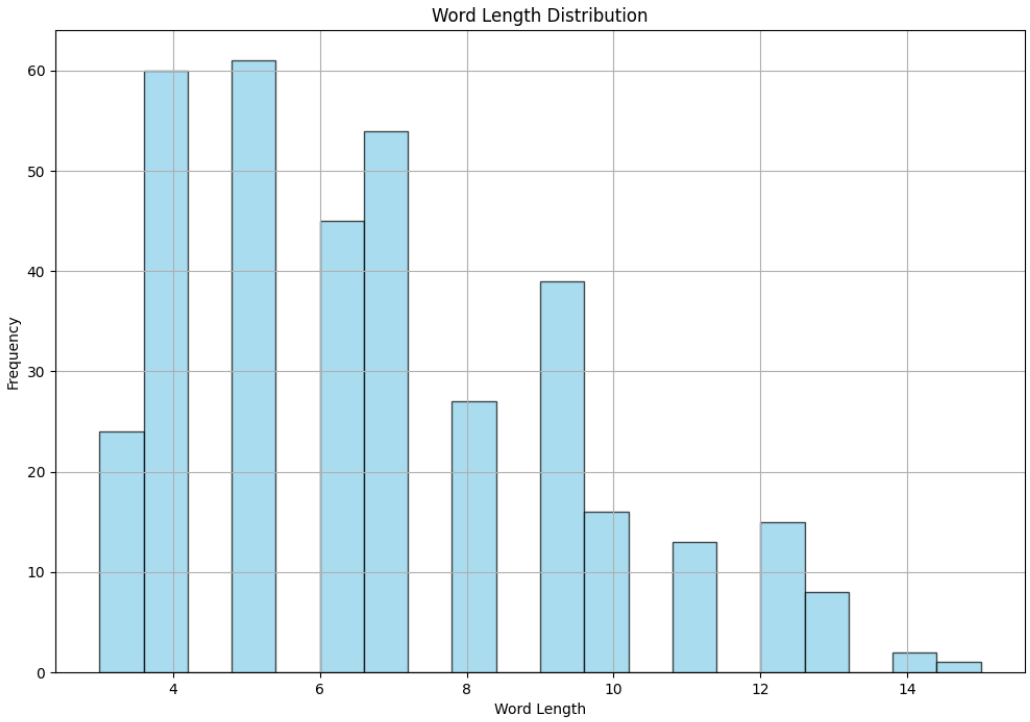
Word Frequency Distribution Histogram

The plot illustrates the frequency distribution of words in the dataset, offering insights into how often each word appears and the overall composition of the text data. The x-axis represents the words, while the y-axis indicates their frequencies. Taller bars signify words that occur more frequently in the dataset. Words like "movie" appear most frequently, with a frequency of 4, followed by "kids," "sense," "cast," and "well," each with a frequency of 3. Other words, including "spy," "franchise," and "problem," have lower frequencies (2). The graph provides insights into the key terms prevalent in the dataset and their relative occurrences.



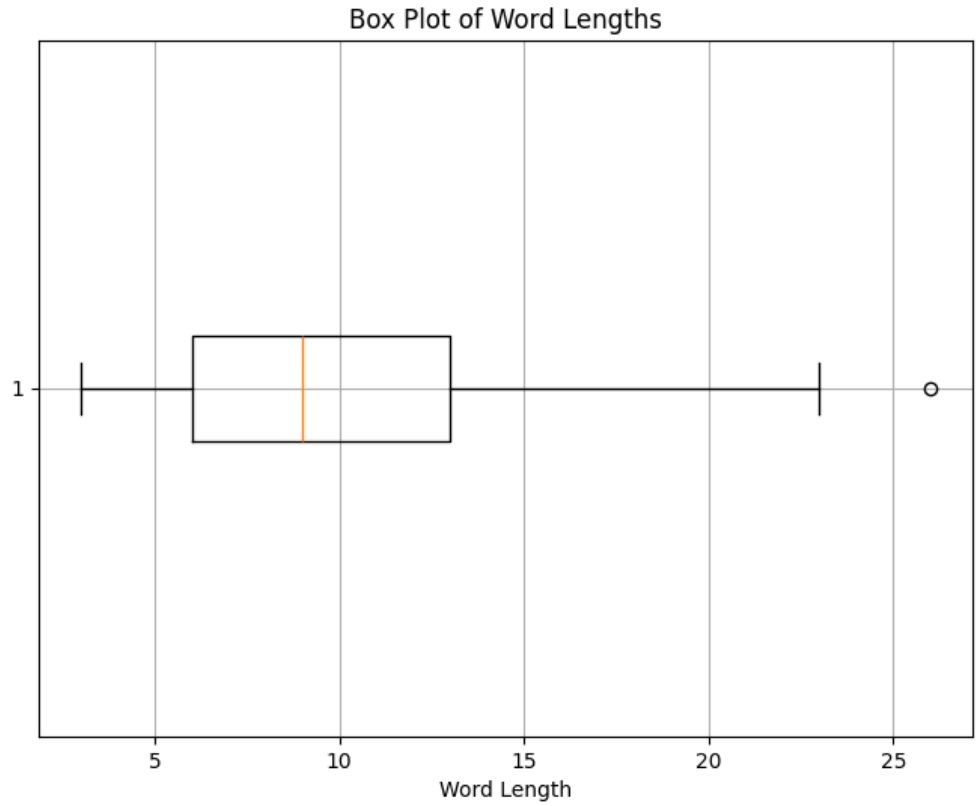
Word length Histogram

This graph represents the distribution of word lengths in the dataset, with the x-axis showing word lengths (in characters) and the y-axis indicating their frequency. Words with 4 and 5 characters are the most frequent, each appearing around 60 times, highlighting the prevalence of shorter words. As word length increases beyond 6 characters, the frequency gradually declines, with words longer than 10 characters occurring much less frequently and very few exceeding 12 characters. The distribution is skewed towards shorter words, reflecting typical linguistic usage patterns and providing insights into the dataset's text composition.



Boxplot for Word Length

The box plot visualizes the distribution of word lengths in the dataset, highlighting central tendency, spread, and outliers. The interquartile range (IQR), represented by the box, captures the middle 50% of word lengths, with the bottom and top edges corresponding to the first (Q1) and third quartiles (Q3), respectively. The median, shown as a line within the box, is around 9–10 characters, while the relatively narrow IQR suggests that most word lengths are clustered within a small range. The whiskers extend to values within 1.5 times the IQR, and a single outlier on the right indicates an unusually long word. Overall, the plot reveals that most words in the dataset are relatively short, with a small presence of longer words.


Box Plot of Word Lengths
1
5
10
15
20
25
Word Length

6. EXPERIMENTS

In the experiments, we used a process called **cross-validation** to assess the performance of different feature sets derived from the data. We applied **multiple model classifiers** including **Naïve Bayes**, **Decision Tree**, **SVM**, and **Random Forest** to compare how well each model could perform with different feature sets, both filtered and unfiltered. The goal was to evaluate how well each feature set can predict sentiment (positive or negative) from the movie reviews.

Cross-Validation Process:

We performed **5-fold cross-validation**, which means the data was divided into 5 equal parts (folds). In each round of the cross-validation, one fold is used as the test set, and the remaining four folds are used to train the model. This process is repeated five times, with each fold serving as the test set once.

Evaluation Metrics:

To measure how well the model performed, we used four key evaluation metrics:

1. **Accuracy** – The percentage of correct predictions made by the model.
2. **Precision** – The percentage of correct positive predictions out of all the predictions made as positive.
3. **Recall** – The percentage of correct positive predictions out of all the actual positives.
4. **F1-Score** – A combined measure of precision and recall, balancing both to give a more overall performance indicator.

Cross-Validation Execution:

The cross-validation process was carried out using functions from the **crossval.py** package. These functions took the data, divided it into batches and ran the evaluation process for each batch. After processing all five batches, we calculated the **mean** (average) scores for accuracy, precision, recall, and F1-score to get an overall performance measure.

Reason: This method ensures that the model is evaluated on different parts of the data multiple times, giving a more reliable estimate of its performance, rather than relying on a single test set. This also helps to prevent overfitting, where a model might perform well on one test set but fail to generalize to new data.

```python
# Importing necessary libraries
import os
import sys
import random
import nltk
from nltk.corpus import stopwords
from collections import defaultdict
import sentiment_read_subjectivity # Custom function to read subjectivity lexicons
import sentiment_read_LIWC_pos_neg_words # Custom function to read LIWC word lists

# Set the directory to the Kaggle data directory
dir = '/Users/nithinkumar/Desktop/FinalProjectData'
os.chdir(dir)
## this code is commented off now, but can be used for sentiment lists

# Initialize the positive, neutral, and negative word lists from the subjectivity lexicon
(positivelist, neutrallist, negativelist)     = sentiment_read_subjectivity.read_subjectivity_three_types('./SentimentLexicons/subjclueslen1-HLTEMNLP05.tff')

# Initialize positive and negative word prefix lists from LIWC
# There is another function 'isPresent' to test if a word's prefix is in the list
(poslist, neglist) = sentiment_read_LIWC_pos_neg_words.read_words()

# Define the feature extraction function
# This function defines features (keywords) of a document for a bag-of-words (BOW) or unigram baseline.
# Each feature is 'V_(keyword)' and is true or false depending on whether that keyword is in the document.
# Define the feature extraction function
def document_features(document, word_features):
    document_words = set(document)  # Convert the document into a set of tokens
    features = {}
    for word in word_features:  # Check if the keyword is in the document
        features['V_{}'.format(word)] = (word in document_words)
    return features
```

```python
# Cross-validation function
# This function takes the number of folds, the feature sets, and the labels.
# It iterates over the folds, using different sections for training and testing in turn.
# It prints the performance for each fold and the average performance at the end.
# Cross-validation function
def cross_validation_PRF(num_folds, featuresets, labels):
    num_labels = len(labels)
    subset_size = int(len(featuresets) / num_folds)
    total_precision_list = [0] * len(labels)
    total_recall_list = [0] * len(labels)
    total_F1_list = [0] * len(labels)
    accuracy_list = []

    for i in range(num_folds):
        test_this_round = featuresets[i * subset_size:][:subset_size]
        train_this_round = featuresets[:i * subset_size] + featuresets[(i + 1) * subset_size:]
        classifier = nltk.NaiveBayesClassifier.train(train_this_round)
        goldlist = [label for (features, label) in test_this_round]
        predictedlist = [classifier.classify(features) for (features, label) in test_this_round]

        precision_list, recall_list, F1_list = eval_measures(goldlist, predictedlist, labels)
        accuracy_this_round = nltk.classify.accuracy(classifier, test_this_round)
        accuracy_list.append(accuracy_this_round)

        for idx in range(len(labels)):
            total_precision_list[idx] += precision_list[idx]
            total_recall_list[idx] += recall_list[idx]
            total_F1_list[idx] += F1_list[idx]

    print('\nAverage Accuracy:', sum(accuracy_list) / num_folds)
    print('\nAverage Precision\tRecall\t\tF1 \tPer Label')
    for idx, lab in enumerate(labels):
        avg_precision = total_precision_list[idx] / num_folds
        avg_recall = total_recall_list[idx] / num_folds
        avg_F1 = total_F1_list[idx] / num_folds
        print(f"{lab}\t{avg_precision:.3f}\t{avg_recall:.3f}\t{avg_F1:.3f}")

    # Calculate precision, recall, and F1 averaged over all rounds for all labels
    precision_list = [tot/num_folds for tot in total_precision_list]
    recall_list = [tot/num_folds for tot in total_recall_list]
    F1_list = [tot/num_folds for tot in total_F1_list]

    print('\nAverage Accuracy : ', sum(accuracy_list)/num_folds)
    # the evaluation measures in a table with one row per label

    print('\nAverage Precision\tRecall\t\tF1 \tPer Label')
    # Macro average over all labels - treats each label equally
```

```
for i, lab in enumerate(labels):
    print(lab, '\t', "{:10.3f}".format(precision_list[i]), \
      "{:10.3f}".format(recall_list[i]), "{:10.3f}".format(F1_list[i]))

# Macro average over all labels - treats each label equally
print('\nMacro Average Precision\tRecall\t\tF1 \tOver All Labels')
print('\t', "{:10.3f}".format(sum(precision_list)/num_labels), \
      "{:10.3f}".format(sum(recall_list)/num_labels), \
      "{:10.3f}".format(sum(F1_list)/num_labels))

# Micro average calculation, weighted by the number of items per label
label_counts = defaultdict(int)  # Default value for any key is int(), which is 0

# Now, you can safely increment any label count without prior initialization
for (doc, lab) in featuresets:
 label_counts[lab] += 1
# make weights compared to the number of documents in featuresets
num_docs = len(featuresets)
label_weights = [(label_counts[lab] / num_docs) for lab in labels]
print('\nLabel Counts', label_counts)
#print('Label weights', label_weights)
# print macro average over all labels
print('Micro Average Precision\tRecall\t\tF1 \tOver All Labels')
precision = sum([a * b for a,b in zip(precision_list, label_weights)])
recall = sum([a * b for a,b in zip(recall_list, label_weights)])
F1 = sum([a * b for a,b in zip(F1_list, label_weights)])
print( '\t', "{:10.3f}".format(precision), \
  "{:10.3f}".format(recall), "{:10.3f}".format(F1))
```

```python
# Function to compute precision, recall, and F1 for each label
# and for any number of labels
# Input: list of gold labels, list of predicted labels (in the same order)
# Output: returns lists of precision, recall, and F1 for each label
# Function to compute precision, recall, and F1 for each label
def eval_measures(gold, predicted, labels):
    recall_list = []
    precision_list = []
    F1_list = []

    for lab in labels:
        TP = FP = FN = 0
        for i, val in enumerate(gold):
            if val == lab and predicted[i] == lab:
                TP += 1
            if val == lab and predicted[i] != lab:
                FN += 1
            if val != lab and predicted[i] == lab:
                FP += 1
        if TP == 0:
            precision = recall = F1 = 0
        else:
            precision = TP / (TP + FP) if (TP + FP) > 0 else 0
            recall = TP / (TP + FN) if (TP + FN) > 0 else 0
            F1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0
        precision_list.append(precision)
        recall_list.append(recall)
        F1_list.append(F1)
    return precision_list, recall_list, F1_list
```

```
# Function to read Kaggle training file, train and test a classifier
# Function to process the Kaggle training file and train/test classifiers
def processkaggle(dirPath, limitStr):
    limit = int(limitStr)
    f = open(os.path.join(dirPath, 'corpus/train.tsv'), 'r')
    phrasedata = []
    for line in f:
        if not line.startswith('Phrase'):
            line = line.strip()
            phrasedata.append(line.split('\t')[2:4])
    random.shuffle(phrasedata)
    phraselist = phrasedata[:limit]

    print(f'Read {len(phrasedata)} phrases, using {len(phraselist)} random phrases')

    phrasedocs = [(nltk.word_tokenize(phrase[0]), int(phrase[1])) for phrase in phraselist]
    all_words_list = [word for (sent, cat) in phrasedocs for word in sent]
    all_words = nltk.FreqDist(all_words_list)
    word_features = [word for (word, count) in all_words.most_common(1500)]
    featuresets = [(document_features(d, word_features), c) for (d, c) in phrasedocs]

    label_list = [c for (d, c) in phrasedocs]
    labels = list(set(label_list))
    num_folds = 5
    cross_validation_PRF(num_folds, featuresets, labels)


"""
commandline interface takes a directory name with kaggle subdirectory for train.tsv
   and a limit to the number of kaggle phrases to use
It then processes the files and trains a kaggle movie review sentiment classifier.

"""
Run | Debug
if __name__ == '__main__':
    if (len(sys.argv) != 3):
        print ('usage: classifyKaggle.py <corpus-dir> <limit>')
        sys.exit(0)
    processkaggle(sys.argv[1], sys.argv[2])
```

CROSS VALIDATION ON FEATURE SETS

Unigram

Filtered

```
 Unigram filtered :

Average Accuracy: 0.53

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.200   0.050   0.080
2       0.594   0.931   0.718
3       0.050   0.067   0.057
4       0.000   0.000   0.000

Average Accuracy :  0.53

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.200      0.050      0.080
2             0.594      0.931      0.718
3             0.050      0.067      0.057
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.169      0.210      0.171

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.382      0.534      0.423
```

Unfiltered

```
 Unigram Unfiltered :

Average Accuracy: 0.55

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.400   0.140   0.204
2       0.607   0.931   0.728
3       0.067   0.067   0.067
4       0.000   0.000   0.000

Average Accuracy :  0.55

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.400      0.140      0.204
2             0.607      0.931      0.728
3             0.067      0.067      0.067
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.215      0.228      0.200

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.440      0.556      0.459
```

Bigram

Filtered

```
 Bigram filtered :

Average Accuracy: 0.49000000000000005

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.000   0.000   0.000
2       0.583   0.905   0.694
3       0.000   0.000   0.000
4       0.000   0.000   0.000

Average Accuracy :  0.49000000000000005

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.000      0.000      0.000
2             0.583      0.905      0.694
3             0.000      0.000      0.000
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.117      0.181      0.139

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.321      0.498      0.382
```

Unfiltered

```
  Bigram Unfiltered :

 Average Accuracy: 0.51

 Average Precision       Recall          F1      Per Label
 0       0.000   0.000   0.000
 1       0.100   0.050   0.067
 2       0.608   0.919   0.722
 3       0.000   0.000   0.000
 4       0.000   0.000   0.000

 Average Accuracy :  0.51

 Average Precision       Recall          F1      Per Label
 0             0.000      0.000      0.000
 1             0.100      0.050      0.067
 2             0.608      0.919      0.722
 3             0.000      0.000      0.000
 4             0.000      0.000      0.000

 Macro Average Precision Recall          F1      Over All Labels
               0.142      0.194      0.158

 Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
 Micro Average Precision Recall          F1      Over All Labels
               0.358      0.517      0.413
```

Pos Tagging

Filtered

```
 Pos filtered :

Average Accuracy: 0.4600000000000001

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.000   0.000   0.000
2       0.569   0.852   0.665
3       0.000   0.000   0.000
4       0.000   0.000   0.000

Average Accuracy :  0.4600000000000001

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.000      0.000      0.000
2             0.569      0.852      0.665
3             0.000      0.000      0.000
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.114      0.170      0.133

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.313      0.469      0.366
```

Unfiltered

```
 Pos Unfiltered :

Average Accuracy: 0.51

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.100   0.050   0.067
2       0.607   0.914   0.721
3       0.000   0.000   0.000
4       0.000   0.000   0.000

Average Accuracy :  0.51

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.100      0.050      0.067
2             0.607      0.914      0.721
3             0.000      0.000      0.000
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.141      0.193      0.158

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.358      0.515      0.413
```

SL Features

Filtered

```
 SL filtered :

Average Accuracy: 0.51

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.200   0.040   0.067
2       0.598   0.920   0.711
3       0.000   0.000   0.000
4       0.000   0.000   0.000

Average Accuracy :  0.51

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.200      0.040      0.067
2             0.598      0.920      0.711
3             0.000      0.000      0.000
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.160      0.192      0.155

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.377      0.516      0.407
```

Unfiltered

```
 SL Unfiltered :

Average Accuracy: 0.51

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.200   0.050   0.080
2       0.613   0.919   0.726
3       0.000   0.000   0.000
4       0.000   0.000   0.000

Average Accuracy :  0.51

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.200      0.050      0.080
2             0.613      0.919      0.726
3             0.000      0.000      0.000
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.163      0.194      0.161

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.385      0.517      0.419
```

LIWC Features

Filtered

```
 LIWC filtered :

Average Accuracy: 0.49000000000000005

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.000   0.000   0.000
2       0.573   0.902   0.690
3       0.000   0.000   0.000
4       0.000   0.000   0.000

Average Accuracy :  0.49000000000000005

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.000      0.000      0.000
2             0.573      0.902      0.690
3             0.000      0.000      0.000
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.115      0.180      0.138

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.315      0.496      0.379
```

Unfiltered

```
 LIWC Unfiltered :

Average Accuracy: 0.51

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.200   0.050   0.080
2       0.599   0.919   0.716
3       0.000   0.000   0.000
4       0.000   0.000   0.000

Average Accuracy :  0.51

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.200      0.050      0.080
2             0.599      0.919      0.716
3             0.000      0.000      0.000
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.160      0.194      0.159

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.377      0.517      0.413
```

Combined SL and LIWC feature

Filtered

```
 Combined SL LIWC filtered:

Average Accuracy: 0.51

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.200   0.040   0.067
2       0.598   0.920   0.711
3       0.000   0.000   0.000
4       0.000   0.000   0.000

Average Accuracy :  0.51

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.200      0.040      0.067
2             0.598      0.920      0.711
3             0.000      0.000      0.000
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.160      0.192      0.155

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.377      0.516      0.407
```

Unfiltered

```
 Combined SL LIWC Unfiltered :

Average Accuracy: 0.51

Average Precision       Recall          F1      Per Label
0       0.000   0.000   0.000
1       0.200   0.050   0.080
2       0.613   0.919   0.726
3       0.000   0.000   0.000
4       0.000   0.000   0.000

Average Accuracy :  0.51

Average Precision       Recall          F1      Per Label
0             0.000      0.000      0.000
1             0.200      0.050      0.080
2             0.613      0.919      0.726
3             0.000      0.000      0.000
4             0.000      0.000      0.000

Macro Average Precision Recall          F1      Over All Labels
              0.163      0.194      0.161

Label Counts defaultdict(<class 'int'>, {2: 55, 1: 24, 3: 15, 4: 3, 0: 3})
Micro Average Precision Recall          F1      Over All Labels
              0.385      0.517      0.419
```

Feature Set	Unigram	Bigram	POS	SL	LIWC	Combined SL-LIWC
Filtered	0.53	0.490	0.460	0.51	0.490	0.51
Unfiltered	0.55	0.51	0.51	0.51	0.51	0.41

Cross-Validation Accuracy Comparison

- Unigram Features: The unfiltered unigram feature set performs slightly better with an accuracy of 0.55 compared to 0.53 for the filtered set. This indicates that filtering does not significantly improve unigram feature performance.
- Bigram Features: The unfiltered bigram feature set (0.51) outperforms the filtered set (0.49). This suggests that bigrams perform similarly with or without filtering.
- POS Features: The unfiltered POS feature set (0.51) achieves better accuracy than the filtered set (0.46), highlighting that filtering may reduce the effectiveness of POS features.
- Sentiment Lexicon (SL) Features: Both the filtered and unfiltered SL feature sets achieve the same accuracy of 0.51, indicating no significant difference between the two.
- LIWC Features: The unfiltered LIWC feature set performs slightly better (0.51) compared to the filtered set (0.49), showing that filtering does not notably improve performance for this feature set.
- Combined SL-LIWC Features: The filtered combined SL-LIWC feature set (0.51) performs better than the unfiltered set (0.41), indicating that filtering improves performance when combining sentiment and LIWC features.

Conclusion

In cross-validation, the unfiltered feature sets tend to perform slightly better than the filtered ones for unigram, bigram, POS, and LIWC features. However, filtering improves the combined SL-LIWC feature set. Overall,

filtering has a mixed impact, with certain feature sets benefiting more from it than others.

Naïve Bayes

The performance of the Naive Bayes classifier was evaluated using different feature sets and data filtering conditions. These feature sets included unigrams, bigrams, part-of-speech (POS) tags, sentiment lexicons (SL), LIWC (Linguistic Inquiry and Word Count) features, and a combination of SL and LIWC features. Accuracy served as the metric for evaluation.

Unigram

Filtered

```
 Unigram filtered :

Accuracy: 60.00%

  | 1 2 3 |
--+-------+
1 |<.>2 . |
2 | .<6>2 |
3 | . .<.>|
--+-------+
(row = reference; col = test)
```

Unfiltered

```
 Unigram Unfiltered :

Accuracy: 60.00%

  | 1 2 3 |
--+-------+
1 |<.>2 . |
2 | 1<6>1 |
3 | . .<.>|
--+-------+
(row = reference; col = test)
```

Bigram

Filtered

```
 Bigram filtered :

Accuracy: 50.00%

  | 0 1 2 3 |
--+---------+
0 |<.>. . . |
1 | .<.>2 . |
2 | 1 .<5>2 |
3 | . . .<.>|
--+---------+
(row = reference; col = test)
```

Unfiltered

```
 Bigram Unfiltered :

Accuracy: 60.00%

  | 1 2 3 |
--+-------+
1 |<.>2 . |
2 | 1<6>1 |
3 | . .<.>|
--+-------+
(row = reference; col = test)
```

Pos Tagging

Filtered

```
 Pos filtered :

Accuracy: 40.00%

  | 0 1 2 3 |
--+---------+
0 |<.>. . . |
1 | .<.>2 . |
2 | 1 2<4>1 |
3 | . . .<.>|
--+---------+
(row = reference; col = test)
```

Unfiltered

```
 Pos Unfiltered :

Accuracy: 60.00%

  | 1 2 3 |
--+-------+
1 |<.>2 . |
2 | 1<6>1 |
3 | . .<.>|
--+-------+
(row = reference; col = test)
```

SL Features

Filtered

```
 SL filtered :

Accuracy: 40.00%

  | 0 1 2 3 |
--+---------+
0 |<.>. . . |
1 | .<.>2 . |
2 | 1 1<4>2 |
3 | . . .<.>|
--+---------+
(row = reference; col = test)
```

Unfiltered

```
 SL Unfiltered :

Accuracy: 50.00%

  | 1 2 3 |
--+-------+
1 |<.>2 . |
2 | 1<5>2 |
3 | . .<.>|
--+-------+
(row = reference; col = test)
```

LIWC Features

Filtered

```
 LIWC filtered :

Accuracy: 40.00%

   | 0 1 2 3 4 |
--+-----------+
0 |<.>. . . . |
1 | .<.>2 . . |
2 | 1 1<4>1 1 |
3 | . . .<.>. |
4 | . . . .<.>|
--+-----------+
(row = reference; col = test)
```

Unfiltered

```
 LIWC Unfiltered :

Accuracy: 50.00%

  | 1 2 3 |
--+-------+
1 |<.>2 . |
2 | 1<5>2 |
3 | . .<.>|
--+-------+
(row = reference; col = test)
```

Combined SL and LIWC Features

Filtered

```
 Combined SL LIWC filtered :

Accuracy: 40.00%

   | 0 1 2 3 |
--+---------+
0 |<.>. . . |
1 | .<.>2 . |
2 | 1 1<4>2 |
3 | . . .<.>|
--+---------+
(row = reference; col = test)
```

Unfiltered

```
 Combined SL LIWC Unfiltered :

Accuracy: 50.00%

   | 1 2 3 |
---+-------+
1 |<.>2 . |
2 | 1<5>2 |
3 | . .<.>|
---+-------+
(row = reference; col = test)
```

Comparison of Accuracies

Feature Set	Unigram	Bigram	POS	SL	LIWC	Combined SL-LIWC
Filtered	60%	50%	40%	40%	40%	40%
Unfiltered	60%	60%	60%	50%	50%	50%

Naïve Bayes Model Performance Analysis: Feature Set Comparison

- Unigram Features: Performance is 60% for both filtered and unfiltered data, suggesting that filtering does not impact the effectiveness of individual word features.
- Bigram Features: Accuracy drops from 60% (unfiltered) to 50% (filtered). Filtering removes valuable context provided by word pairs, leading to lower performance.
- POS Features: Accuracy drops significantly from 60% (unfiltered) to 40% (filtered). Filtering likely removes important syntactic information, impacting classification.
- Sentiment Lexicon (SL) Features: Accuracy increases from 40% (filtered) to 50% (unfiltered). Filtering removes key sentiment-bearing words, reducing the model's effectiveness.
- LIWC Features: Similar to SL features, performance improves from 40% (filtered) to 50% (unfiltered) by retaining more sentiment-related words.

- Combined SL-LIWC Features: Unfiltered data performs better at 50%, compared to 40% for filtered data, indicating that combining both features benefits from retaining more information.

Unfiltered data generally provides better performance across all feature sets. It could be possibe that filtering removes valuable information, particularly for contextual and sentiment-related features, which leads to lower classification accuracy.

Decision tree

The Decision Tree classifier was evaluated for sentiment analysis using different feature sets and data filtering conditions. The feature sets tested included unigram, bigram, POS, SL, LIWC, and a combination of SL and LIWC, with both filtered and unfiltered data being assessed.

Filtered

```
===== for filtered =====

 Unigram filtered :
Classifier: Decision Tree
Accuracy: 80.00%

 Bigram filtered :
Classifier: Decision Tree
Accuracy: 80.00%

 Pos filtered :
Classifier: Decision Tree
Accuracy: 50.00%

 SL filtered :
Classifier: Decision Tree
Accuracy: 60.00%

 LIWC filtered :
Classifier: Decision Tree
Accuracy: 50.00%

 Combined SL LIWC filtered :
Classifier: Decision Tree
Accuracy: 60.00%
```

Unfiltered

```
 Unigram Unfiltered :
Classifier: Decision Tree
Accuracy: 70.00%

 Bigram Unfiltered :
Classifier: Decision Tree
Accuracy: 20.00%

 Pos Unfiltered :
Classifier: Decision Tree
Accuracy: 50.00%

 SL Unfiltered :
Classifier: Decision Tree
Accuracy: 20.00%

 LIWC Unfiltered :
Classifier: Decision Tree
Accuracy: 40.00%

 Combined SL LIWC Unfiltered :
Classifier: Decision Tree
Accuracy: 20.00%
```

Feature Set	Unigram	Bigram	POS	SL	LIWC	Combined SL-LIWC
Filtered	80%	80%	50%	60%	50%	60%
Unfiltered	70%	20%	50%	20%	40%	20%

Decision Tree Model Performance Analysis: Feature Set Comparison

- Unigram Features: The filtered unigram feature set performs well at 80%, compared to 70% for unfiltered data. Filtering appears to improve performance by removing noise and irrelevant words.
- Bigram Features: Both filtered and unfiltered bigram sets perform similarly at 80%. This suggests that bigram context is robust enough to handle filtering without a significant drop in accuracy.
- POS Features: Performance is the same at 50% for both filtered and unfiltered data, indicating that filtering does not significantly impact the importance of part-of-speech information.
- Sentiment Lexicon (SL) Features: The filtered SL feature set performs at 60%, significantly better than the 20% for unfiltered. Filtering seems to help by focusing on key sentiment-bearing words, improving classification accuracy.
- LIWC Features: Filtered LIWC features outperform unfiltered ones at 50% vs. 40%. Filtering enhances performance by retaining words that are more relevant to the analysis.

- Combined SL-LIWC Features: The filtered combined feature set performs better at 60%, compared to 20% for the unfiltered data. The combination of filtered sentiment lexicon and LIWC features provides more meaningful insights, improving model accuracy.

Conclusion

For the Decision Tree model, filtering has a significant positive impact, especially for SL and combined SL-LIWC features. Unlike Naïve Bayes, which performed better with unfiltered data, the Decision Tree model benefits from data preprocessing, which helps eliminate irrelevant information and focuses on important features.

SVM Classifier

The SVM classifier was evaluated for sentiment analysis using different feature sets and data filtering conditions. The feature sets included unigram, bigram, POS, SL, LIWC, and a combined SL-LIWC set.

Filtered

```
===== for filtered =====

 Unigram filtered :
Classifier: SVM
Accuracy: 50.00%

 Bigram filtered :
Classifier: SVM
Accuracy: 80.00%

 Pos filtered :
Classifier: SVM
Accuracy: 50.00%

 SL filtered :
Classifier: SVM
Accuracy: 30.00%

 LIWC filtered :
Classifier: SVM
Accuracy: 50.00%

 Combined SL LIWC filtered :
Classifier: SVM
Accuracy: 30.00%
```

Unfiltered

```
 Unigram Unfiltered :
Classifier: SVM
Accuracy: 60.00%

 Bigram Unfiltered :
Classifier: SVM
Accuracy: 50.00%

 Pos Unfiltered :
Classifier: SVM
Accuracy: 40.00%

 SL Unfiltered :
Classifier: SVM
Accuracy: 50.00%

 LIWC Unfiltered :
Classifier: SVM
Accuracy: 50.00%

 Combined SL LIWC Unfiltered :
Classifier: SVM
Accuracy: 50.00%
```

Feature Set	Unigram	Bigram	POS	SL	LIWC	Combined SL-LIWC
Filtered	50%	80%	50%	30%	50%	30%
Unfiltered	60%	50%	40%	50%	50%	50%

SVM Classifier Performance Analysis: Feature Set Comparison

- Unigram Features: The unfiltered unigram feature set performs slightly better (60%) compared to the filtered set (50%). This suggests that filtering does not significantly improve the performance of individual word features for SVM classification.
- Bigram Features: Filtering the bigram feature set leads to a substantial improvement, with an 80% accuracy compared to 50% for unfiltered data. This

indicates that bigrams provide better context when filtered, which enhances the model's ability to capture important patterns.

- POS Features: Both filtered and unfiltered POS feature sets perform the same at 50%. This suggests that part-of-speech features are not greatly affected by filtering, and their contribution to the SVM model remains consistent.
- Sentiment Lexicon (SL) Features: The unfiltered SL feature set performs better at 50%, compared to 30% for the filtered set. Filtering reduces the number of sentiment-related words, which appears to harm performance for SVM classification in this case.
- LIWC Features: Both filtered and unfiltered LIWC feature sets perform similarly at 50%. Filtering does not significantly impact the performance, suggesting that LIWC features are relatively robust to filtering.
- Combined SL-LIWC Features: The filtered combined SL-LIWC feature set performs worse at 30% compared to the unfiltered set (50%). This suggests that filtering may eliminate crucial sentiment-related words, reducing the model's ability to classify accurately when combining both sentiment and LIWC features.

Conclusion

For the SVM classifier, the filtered bigram feature set significantly improves performance, while other feature sets, especially sentiment lexicon and combined SL-LIWC, perform better without filtering. This highlights the importance of context provided by bigrams and the potential drawbacks of overly aggressive filtering for sentiment-related features in SVM classification.

Random Forest Classifier

The Random forest classifier was evaluated for sentiment analysis using different feature sets and data filtering conditions. The feature sets included unigram, bigram, POS, SL, LIWC, and a combined SL-LIWC set.

Filtered

```
===== for filtered =====

 Unigram filtered :
Classifier: Random Forest
Accuracy: 80.00%

 Bigram filtered :
Classifier: Random Forest
Accuracy: 80.00%

 Pos filtered :
Classifier: Random Forest
Accuracy: 60.00%

 SL filtered :
Classifier: Random Forest
Accuracy: 70.00%

 LIWC filtered :
Classifier: Random Forest
Accuracy: 50.00%

 Combined SL LIWC filtered :
Classifier: Random Forest
Accuracy: 70.00%
```

Unfiltered

```
 Unigram Unfiltered :
Classifier: Random Forest
Accuracy: 80.00%

 Bigram Unfiltered :
Classifier: Random Forest
Accuracy: 50.00%

 Pos Unfiltered :
Classifier: Random Forest
Accuracy: 50.00%

 SL Unfiltered :
Classifier: Random Forest
Accuracy: 40.00%

 LIWC Unfiltered :
Classifier: Random Forest
Accuracy: 40.00%

 Combined SL LIWC Unfiltered :
Classifier: Random Forest
Accuracy: 40.00%
```

Feature Set	Unigram	Bigram	POS	SL	LIWC	Combined SL-LIWC
Filtered	80%	80%	60%	70%	50%	70%
Unfiltered	80%	50%	50%	40%	40%	40%

Random Forest Classifier Performance Analysis: Feature Set Comparison

- **Unigram Features:** Both filtered and unfiltered unigram feature sets achieve the same accuracy of 80%. This indicates that unigram features are highly effective for the Random Forest classifier, and filtering does not improve or degrade performance in this case.
- **Bigram Features:** Filtering improves the performance of the bigram feature set, with an 80% accuracy compared to 50% for the unfiltered set. This suggests that bigrams, when filtered, help the model capture relevant context and improve classification accuracy.
- **POS Features:** The filtered POS feature set outperforms the unfiltered set (60% vs. 50%). Filtering enhances the usefulness of POS features for the Random Forest classifier, allowing it to focus on more meaningful parts of speech.
- **Sentiment Lexicon (SL) Features:** The filtered SL feature set performs better at 70%, compared to 40% for the unfiltered set. This shows that removing unnecessary or less relevant sentiment words through filtering helps improve performance for the Random Forest model.
- **LIWC Features:** The filtered LIWC feature set performs better at 50%, while the unfiltered set scores 40%. This indicates that filtering may help focus on the most relevant LIWC categories, enhancing the Random Forest model's ability to classify effectively.
- **Combined SL-LIWC Features:** Filtering improves the performance of the combined SL-LIWC feature set, with an accuracy of 70% compared to 40% for the unfiltered set. This suggests that combining filtered sentiment and LIWC features allows the model to capture more relevant patterns, improving its classification ability.

Conclusion

For the Random Forest classifier, filtering generally improves the performance of bigrams, POS, SL, and combined SL-LIWC features. Unigram features perform equally well with and without filtering. This highlights the importance of filtering for improving

the performance of certain feature sets, particularly bigrams, sentiment lexicons, and combined features in Random Forest classification.

OBSEVATIONS

Comparison of Feature Sets and Models

1. Feature Set Performance Comparison:

- **Unigram Features:**
    - **Filtered:** 60% (Naïve Bayes, Random Forest), 80% (Decision Tree)
    - **Unfiltered:** 60% (Naïve Bayes, Random Forest), 70% (Decision Tree, SVM)
    - **Conclusion:** Unigram features consistently show a high level of performance, especially for the Decision Tree model (80% filtered). There's little improvement with filtering for Naïve Bayes and Random Forest.
- **Bigram Features:**
    - **Filtered:** 50% (Naïve Bayes, SVM), 80% (Random Forest, Decision Tree)
    - **Unfiltered:** 60% (Naïve Bayes), 50% (SVM), 80% (Decision Tree, Random Forest)
    - **Conclusion:** Bigrams are more effective for Decision Tree and Random Forest models, especially when unfiltered (80% accuracy).
- **POS Features:**
    - **Filtered:** 40% (Naïve Bayes, SVM), 50% (Random Forest, Decision Tree)
    - **Unfiltered:** 50% (Naïve Bayes, SVM, Random Forest), 60% (Decision Tree)
    - **Conclusion:** POS features show relatively low performance overall, with no significant difference between filtered and unfiltered versions.
- **Sentiment Lexicon (SL) Features:**
    - **Filtered:** 40% (Naïve Bayes, SVM), 60% (Random Forest, Decision Tree)
    - **Unfiltered:** 50% (Naïve Bayes, SVM), 40% (Random Forest), 50% (Decision Tree)
    - **Conclusion:** SL features perform well in Decision Tree and Random Forest models, with filtering slightly improving performance in these models.

- **LIWC Features:**
    - **Filtered:** 40% (Naïve Bayes, SVM), 50% (Random Forest), 60% (Decision Tree)
    - **Unfiltered:** 50% (Naïve Bayes, SVM), 40% (Random Forest), 50% (Decision Tree)
    - **Conclusion:** LIWC features perform comparably to SL features, with Decision Tree performing best for both filtered and unfiltered versions.
- **Combined SL-LIWC Features:**
    - **Filtered:** 40% (Naïve Bayes, SVM), 60% (Random Forest, Decision Tree)
    - **Unfiltered:** 50% (Naïve Bayes, SVM), 40% (Random Forest), 50% (Decision Tree)
    - **Conclusion:** The combined SL-LIWC feature set performs best in Decision Tree and Random Forest models, especially with filtering.

2. Model Performance Comparison:

- **Naïve Bayes:**
    - **Filtered:** Generally performs well with unigram features (60%) but does not excel with bigram, POS, or sentiment-related features.
    - **Unfiltered:** Performs consistently across unigrams, bigrams, and sentiment features with accuracies around 60%.
- **Decision Tree:**
    - **Filtered:** Shows strong performance with unigram, bigram, and sentiment features, especially unigram (80%) and bigram (80%).
    - **Unfiltered:** Performs best with unigram (70%) and bigram (80%) features, showing a consistent and high level of accuracy across different feature sets.
- **SVM:**
    - **Filtered:** Generally struggles with all feature sets, with the highest performance in bigrams (50%).
    - **Unfiltered:** Performs best with unigram and sentiment features (50%) but underperforms with other feature sets.
- **Random Forest:**

- **Filtered:** Outperforms Naïve Bayes and SVM in several feature sets, with particularly strong performance in unigrams and bigrams (80%).
- **Unfiltered:** Performs well in unigrams (80%) and bigrams (50%), showing that it can handle both filtered and unfiltered features with similar success.

3. Cross-Validation Accuracy:

- **Filtered:** The accuracy remains relatively low across most models and feature sets, with no significant improvements over unfiltered data for certain models (especially Naïve Bayes and SVM).
- **Unfiltered:** Shows higher accuracy in most cases compared to filtered data, especially with Naïve Bayes, Decision Tree, and Random Forest models.

Best Performing Feature Sets:

Unigram Features generally perform well across all models, particularly in Decision Tree and Random Forest classifiers.

Bigram Features perform best for Decision Tree and Random Forest, especially in unfiltered conditions.

Combined SL-LIWC features perform well with Decision Tree, but the combination shows mixed results across other models.

Best Performing Models:

Decision Tree performs best overall, particularly with unigram and bigram features, and also shows high accuracy when combined with sentiment lexicon (SL) and LIWC features (both filtered and unfiltered).

Random Forest also performs well across all feature sets, performing similarly to Decision Tree but with slight variations in feature effectiveness.

Naïve Bayes tends to perform similarly across all feature sets but struggles with sentiment-related features.

SVM performs the worst in most cases, particularly with filtered data.

LESSONS LEARNED:

Feature Set Selection: The performance of unigram and bigram features is consistently strong, suggesting that simple n-grams can often capture useful patterns in text classification tasks.

- **Model Sensitivity to Features:** Different models perform better with different feature sets. Decision Tree and Random Forest perform exceptionally well with n-grams, while Naïve Bayes and SVM may struggle with more complex features like sentiment lexicons or LIWC.
- **Filtering Impact:** Filtering does not always improve performance and sometimes even reduces accuracy, especially in models like Naïve Bayes and SVM. However, it can benefit certain feature sets like combined SL-LIWC in some cases.
- **Cross-Validation Results:** Unfiltered data tends to perform better overall, but filtering can sometimes add value for specific feature sets and models.

OVERALL CONCLUSION:
In conclusion, the goal was to evaluate how well each feature set could predict sentiment (positive or negative) from the movie reviews. Based on the results, **Decision Tree** and **Random Forest** classifiers performed the best overall, achieving higher accuracy compared to other models. The **unigram** and **bigram** feature sets proved to be the most effective in capturing sentiment, while **Naïve Bayes** and **SVM** classifiers showed lower performance, especially when using sentiment lexicon (SL) and LIWC-based features. **Filtering** the data was beneficial in certain cases, but not universally applicable, as unfiltered data often yielded higher accuracy than filtered data for several models. This highlights the importance of experimenting with different preprocessing techniques and classifiers to determine the best combination for sentiment analysis.

.

CHALLENGES

- The dataset used in this analysis was significantly more complex and extensive compared to those typically encountered in class assignments. As a result, the process of exploring and understanding the dataset, as well as fully grasping the intricacies of the problem statement, required considerable time and effort. This involved delving deeply into the data to identify patterns, relationships, and potential challenges, ensuring a thorough comprehension of the task before proceeding with modeling and evaluation.

- Since the project involved custom-made functions stored in separate files, integrating and compiling the overall code to ensure seamless access to these functions posed a significant challenge. Additionally, managing and correctly linking the required CSV files further added to the complexity, as it required careful organization and attention to file paths and dependencies to avoid errors during execution.

- Initially, the models yielded quite low accuracy, which indicated that the preprocessing steps were insufficient or ineffective. This prompted us to revisit and refine the preprocessing pipeline. By systematically addressing these issues and ensuring the data was prepared appropriately, we were able to improve the quality of the input features, which subsequently led to better model performance.

TASK DISTRIBUTION:

Resham Bahira:

1. Collecting Data and Preprocessing
2. Generating Feature sets
3. Data Visualization
4. Writing Final Report

Nithin Kumar Hadhge Girish Kumar:

5. Generating Feature sets
6. Saving data to csv files

7. Experiments which include cross validation and generating other models
8. Compiling the final code