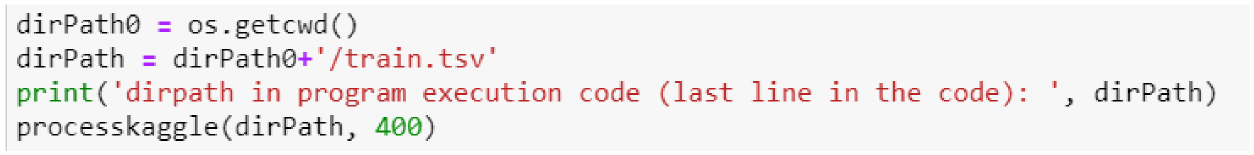
1. Dataset

a. Data Description

For this final project, the Rotten Tomatoes movie review dataset is a corpus of movie reviews used for sentiment analysis, collected by Pang and Lee from Cornell University. Sentiment treebanks from Socher et al. were applied to the dataset to create fine-grained labels for all parsed phrases in the corpus. The train dataset consists of 156060 rows and 4 columns. The columns are labeled PhraseID, SentenceID, Phrase, and Sentiment. In the Sentiment column on a scale of five values: 0 = negative, 1 = somewhat negative, 2 = neutral, 3 = somewhat positive, 4= positive. A directory path (Figure 1) is mapped out on retrieving the train.tsv dataset into a python editor. `processkaggle(dirPath, 400)` set a limit on only 400 Kaggle movie phrases to use, process the phrases, and train all four features/classifiers.

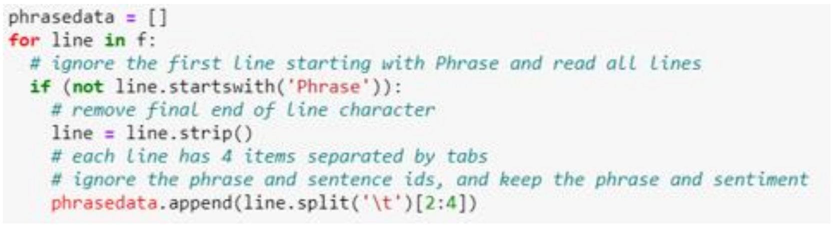


```
dirPath0 = os.getcwd()
dirPath = dirPath0+'/train.tsv'
print('dirpath in program execution code (last line in the code): ', dirPath)
processkaggle(dirPath, 400)
```

Figure 1: Directory Path and Processkaggle Setup

b. Data Pre-processing

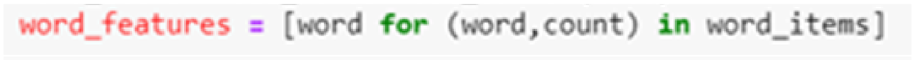
The train dataset is converted to phrasedata (Figure 2) to pull only the phrase and sentiment columns. Then randomize the phrasedata, and create a phraselist of 400 randomized phrases. Then, a phrasedocs generate a list of phrase documents as (list of words, label).



```
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 the phrase and sentence ids, and keep the phrase and sentiment
    phrasedata.append(line.split('\t')[2:4])
```

Figure 2: Phrasedata

The phrase in the phraselist is tokenized, and then all words are converted to lower case. Then, generated all_words_list, created an all_words based on `nltk.FreqDist(all_words_list)`, and lastly word_items based on the top 1500 words. Importantly, word_features (Figure 3) is set up to be applied for classification activities.



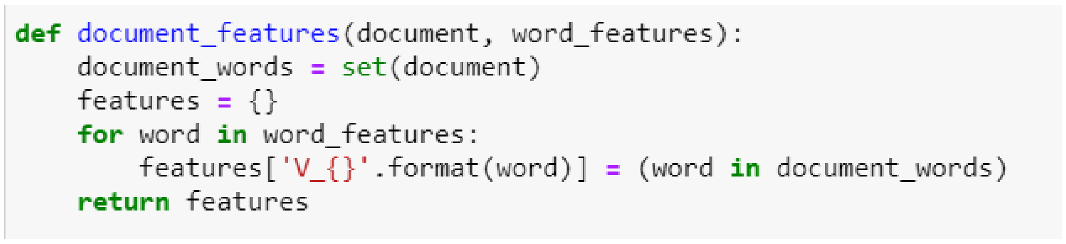
```
word_features = [word for (word,count) in word_items]
```

Figure 3: word_features

2. Features

a. document_features

This document_features (Figure 4) is a function that defines the keywords of the corpus for a bag-of-words or unigram. The word feature labels V_(keyword) will be determined whether true or false if the keyword is in the corpus.

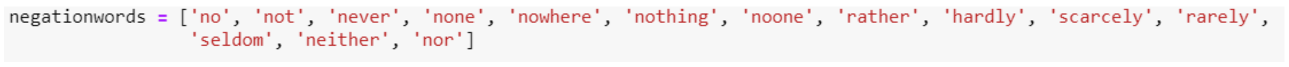


```
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    return features
```

Figure 4: Document_features

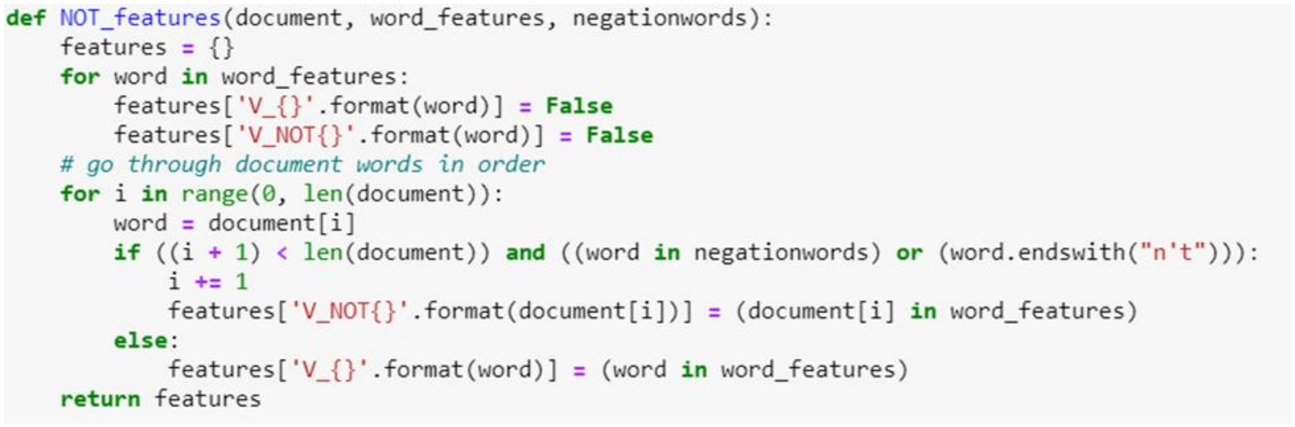
b. NOT_features

NOT_features is a function that defines the negationwords (Figure 5) and the keywords of the corpus for a bag-of-words or unigram. The feature labels V_(keyword) and V_NOT (keyword) will be determined whether true or false if the keyword is in the corpus (Figure 6).



```
negationwords = ['no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone', 'rather', 'hardly', 'scarcely', 'rarely',
                 'seldom', 'neither', 'nor']
```

Figure 5: Negation

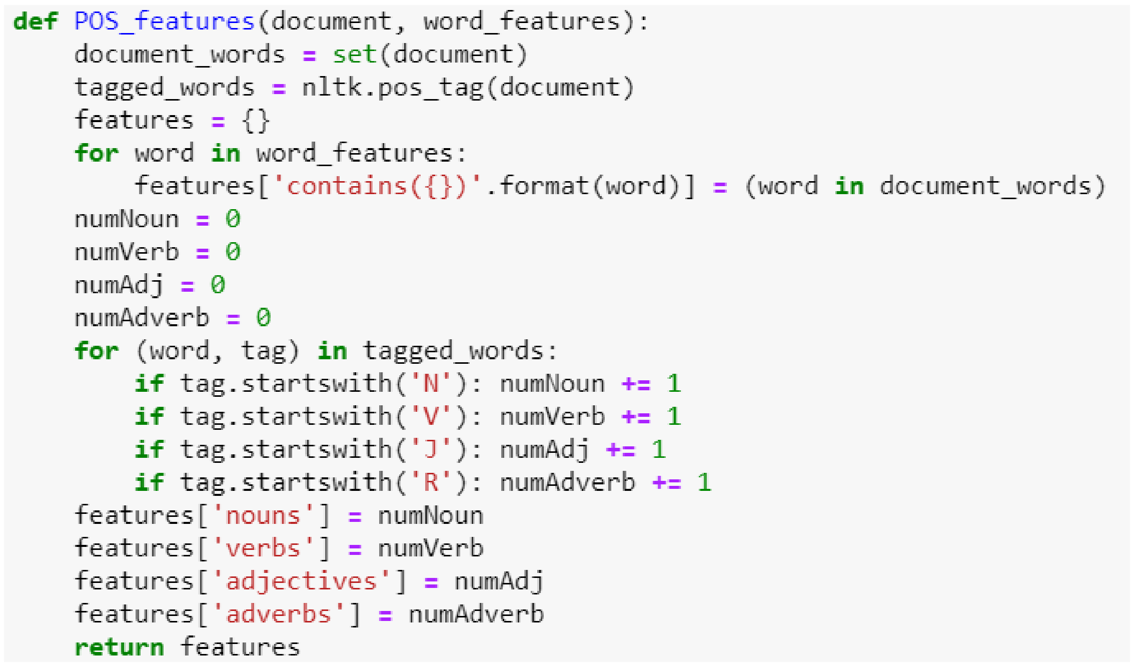


```python
def NOT_features(document, word_features, negationwords):
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = False
        features['V_NOT{}'.format(word)] = False
    # go through document words in order
    for i in range(0, len(document)):
        word = document[i]
        if ((i + 1) < len(document)) and ((word in negationwords) or (word.endswith("n't"))):
            i += 1
            features['V_NOT{}'.format(document[i])] = (document[i] in word_features)
        else:
            features['V_{}'.format(word)] = (word in word_features)
    return features
```

Figure 6: NOT_features

c. POS_features

POS_features (Figure 7) is a function that defines the various types of word tags (noun, verb, adjective, and adverb) with the word_features. The POS_feature counts the number of tagged_word is in the corpus.

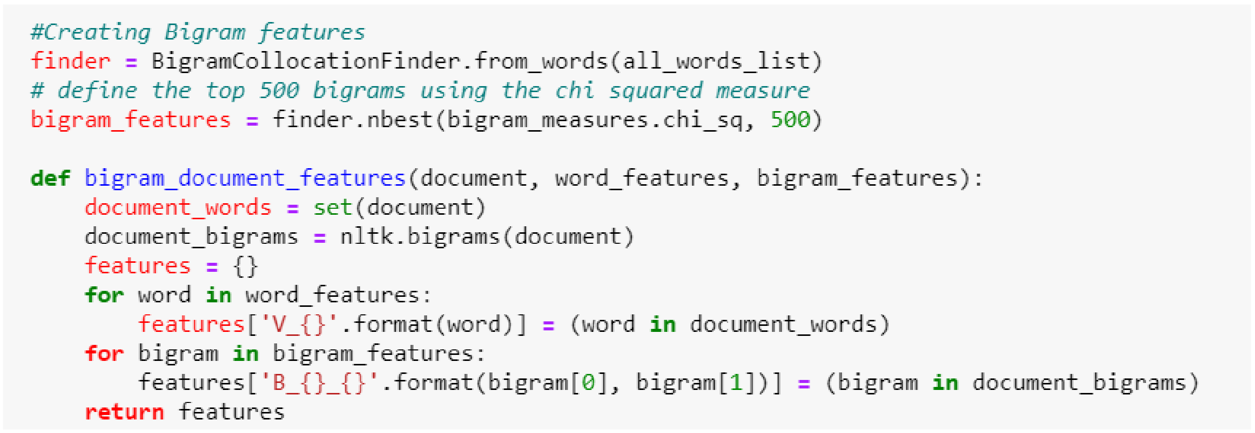


```python
def POS_features(document, word_features):
    document_words = set(document)
    tagged_words = nltk.pos_tag(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    numNoun = 0
    numVerb = 0
    numAdj = 0
    numAdverb = 0
    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
    features['nouns'] = numNoun
    features['verbs'] = numVerb
    features['adjectives'] = numAdj
    features['adverbs'] = numAdverb
    return features
```

Figure 7: POS_features

d. Bigram_document_features

The Bigram_document_features (Figure 8) is a function that defines the bigram_features and word_features. The Bigram_feature is a function that uses the chi-squared measure to get bigrams. Additionally, the nbest function is used to provide the highest scoring bigrams.

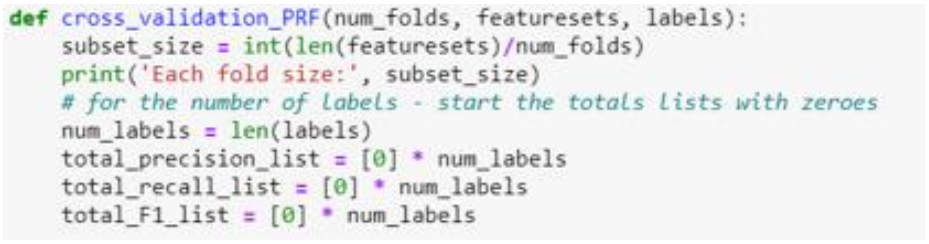


```
#Creating Bigram features
finder = BigramCollocationFinder.from_words(all_words_list)
# define the top 500 bigrams using the chi squared measure
bigram_features = finder.nbest(bigram_measures.chi_sq, 500)

def bigram_document_features(document, word_features, bigram_features):
    document_words = set(document)
    document_bigrams = nltk.bigrams(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    for bigram in bigram_features:
        features['B_{}_{}'.format(bigram[0], bigram[1])] = (bigram in document_bigrams)
    return features
```

Figure 8: Bigram_document_features

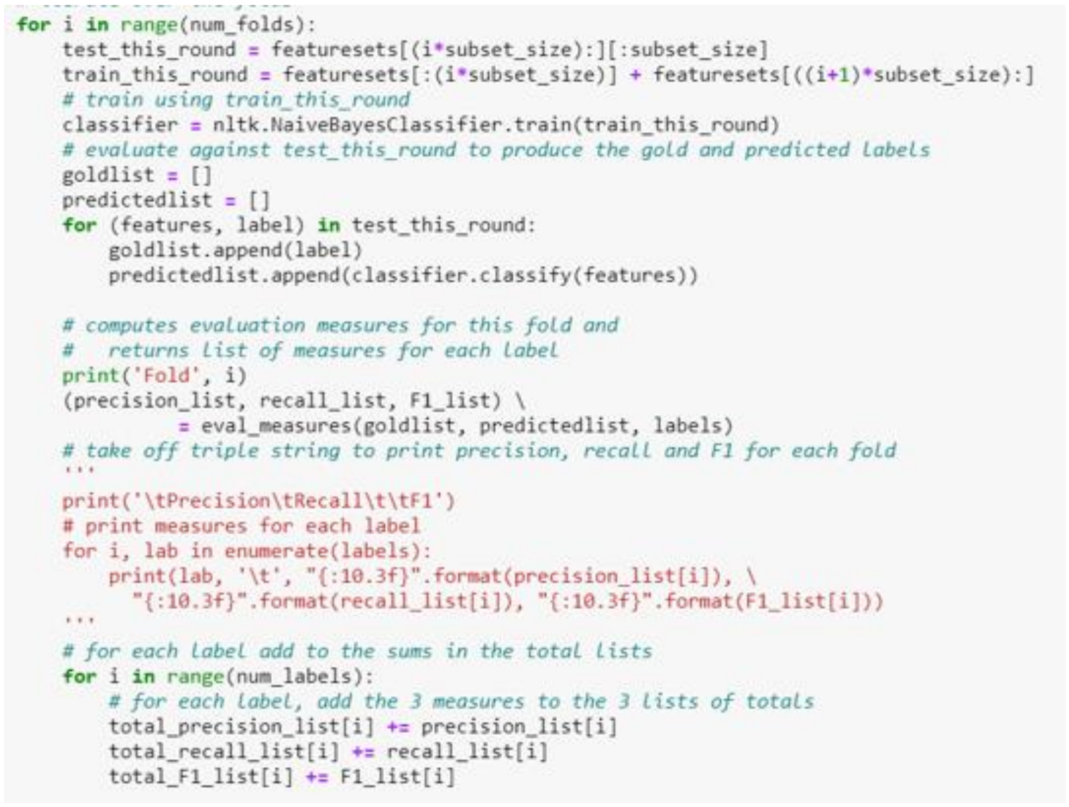
e. Cross-Validation

The cross-validation (Figure 9) is a function that sets 10 folds for this project, the four feature sets, and the labels. Then run the cross-validation, using different sections for training and testing in 10 folds (Figure 10) for each nltk.NaiveBayesClassifier. A goldlist = [] and predictedlist =[] are generated. Also, the function computes the measurement of precision_list, recall_list, and F1_list.



```
def cross_validation_PRF(num_folds, featuresets, labels):
    subset_size = int(len(featuresets)/num_folds)
    print('Each fold size:', subset_size)
    # for the number of labels - start the totals lists with zeroes
    num_labels = len(labels)
    total_precision_list = [0] * num_labels
    total_recall_list = [0] * num_labels
    total_F1_list = [0] * num_labels
```

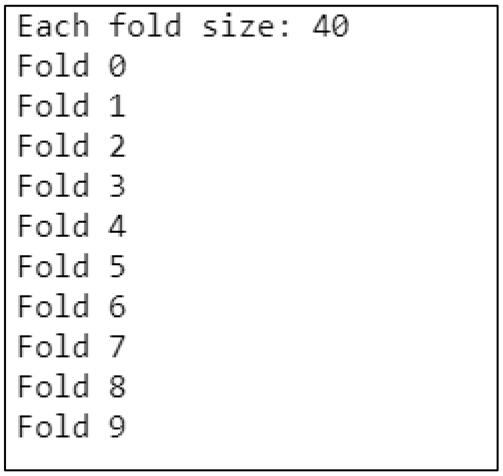
Figure 9: cross_validation_PRE



```
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):]
    # train using train_this_round
    classifier = nltk.NaiveBayesClassifier.train(train_this_round)
    # evaluate against test_this_round to produce the gold and predicted labels
    goldlist = []
    predictedlist = []
    for (features, label) in test_this_round:
        goldlist.append(label)
        predictedlist.append(classifier.classify(features))

    # computes evaluation measures for this fold and
    #   returns list of measures for each label
    print('Fold', i)
    (precision_list, recall_list, F1_list) \
              = eval_measures(goldlist, predictedlist, labels)
    # take off triple string to print precision, recall and F1 for each fold
    '''
    print('\tPrecision\tRecall\t\tF1')
    # print measures for each label
    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]))
    '''
    # for each label add to the sums in the total lists
    for i in range(num_labels):
        # for each label, add the 3 measures to the 3 lists of totals
        total_precision_list[i] += precision_list[i]
        total_recall_list[i] += recall_list[i]
        total_F1_list[i] += F1_list[i]
```

Figure 10: Fold and Measurement in cross_validation_PRE

Since the phrase limit for this project is set at 400, 10-fold cross-validation and 40 phrases are processed in each fold (Figure 11).



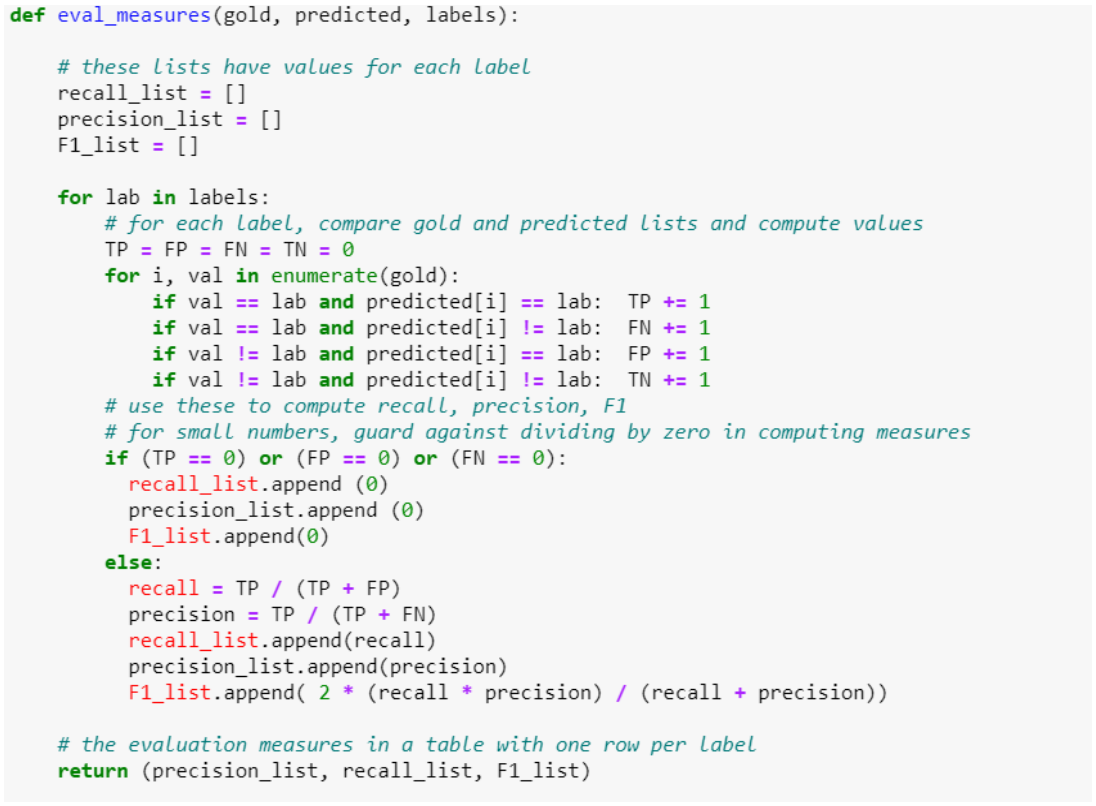
```
Each fold size: 40
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Fold 6
Fold 7
Fold 8
Fold 9
```

Figure 11: Output from 10-fold cross-validation

f. Evaluation Measurements: Precision, Recall, and F1

The eval_ measures function is on how accurately each classifier (Figure 12) was generated from this experiment. This function identifies the number of true positive (TP), false negative (FN), false positive (FP), and true negative (TN) for each sentiment label. Then, the following calculations are generated to compare among the classifiers for their accuracy for each sentiment label:

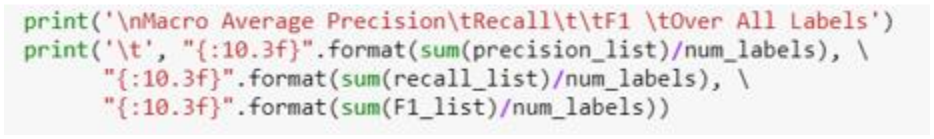
1. Recall is based on TP / (TP + FP), which is the percentage of actual yes answers that are right.
2. Precision is based on TP / (TP + FN), which is the percentage of predicted yes answers that are right.
3. F1 is based on (2 * (recall * precision) / (recall + precision)), which is the combined harmonic mean of recall and precision.



```
def eval_measures(gold, predicted, labels):

    # these lists have values for each label
    recall_list = []
    precision_list = []
    F1_list = []

    for lab in labels:
        # for each label, compare gold and predicted lists and compute values
        TP = FP = FN = TN = 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 val != lab and predicted[i] != lab:  TN += 1
        # use these to compute recall, precision, F1
        # for small numbers, guard against dividing by zero in computing measures
        if (TP == 0) or (FP == 0) or (FN == 0):
          recall_list.append (0)
          precision_list.append (0)
          F1_list.append(0)
        else:
          recall = TP / (TP + FP)
          precision = TP / (TP + FN)
          recall_list.append(recall)
          precision_list.append(precision)
          F1_list.append( 2 * (recall * precision) / (recall + precision))

    # the evaluation measures in a table with one row per label
    return (precision_list, recall_list, F1_list)
```

Figure 12: eval_measures

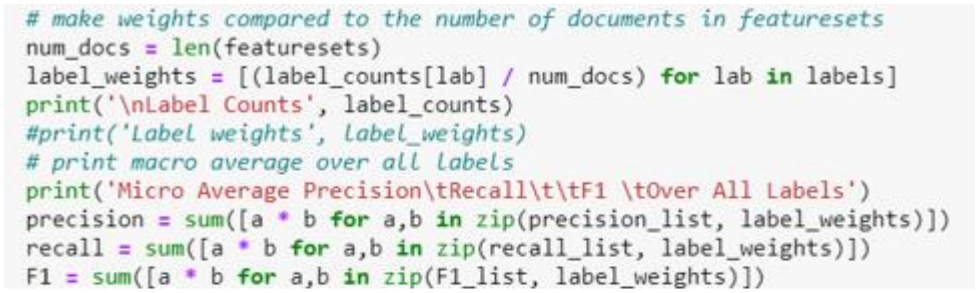
Figure 13 is the function to calculate the overall accuracy based on Marco average precision, recall, and F1and micro average. The Macro average is the sum of precision/recall/F1_list (labeled average precision) / num_labels.



```
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))
```

Figure 13: Macro Average Precision/Recall/F1

Figure 14 shows the function that calculates the overall accuracy by incorporating the weight per sentiment label to reasonably determine the Micro average precision, recall, and F1. The Micro average is the sum of precision/recall/F1_list (labeled average precision) / num_labels with the weights included.



```
# 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)])
```

Figure 14: Weighted Micro Average Precision/Recall/F1

3. Experiment

a. Data Distribution

Four hundred rows of Kaggle movie review's train data are pulled from the train.tsv. The rows are randomized, tokenized, lowercased, and restructured to create word_items. The word_items are applied for cross-validation and feature/classifier to evaluate the best features for the data. Figure 15 shows how the data was distributed among the 400 randomized rows within the 5 sentiment labels. The negative (0) and positive (4) labels have the lowest percentage counts ~ 6 – 7%. The neutral (2) sentiment label has the highest percentage count of 50%. Then, the somewhat positive (3) and negative (1) sentiment labels have ~ percentage counts of 17-20%. This initial view of the data indicated the data is not balanced.

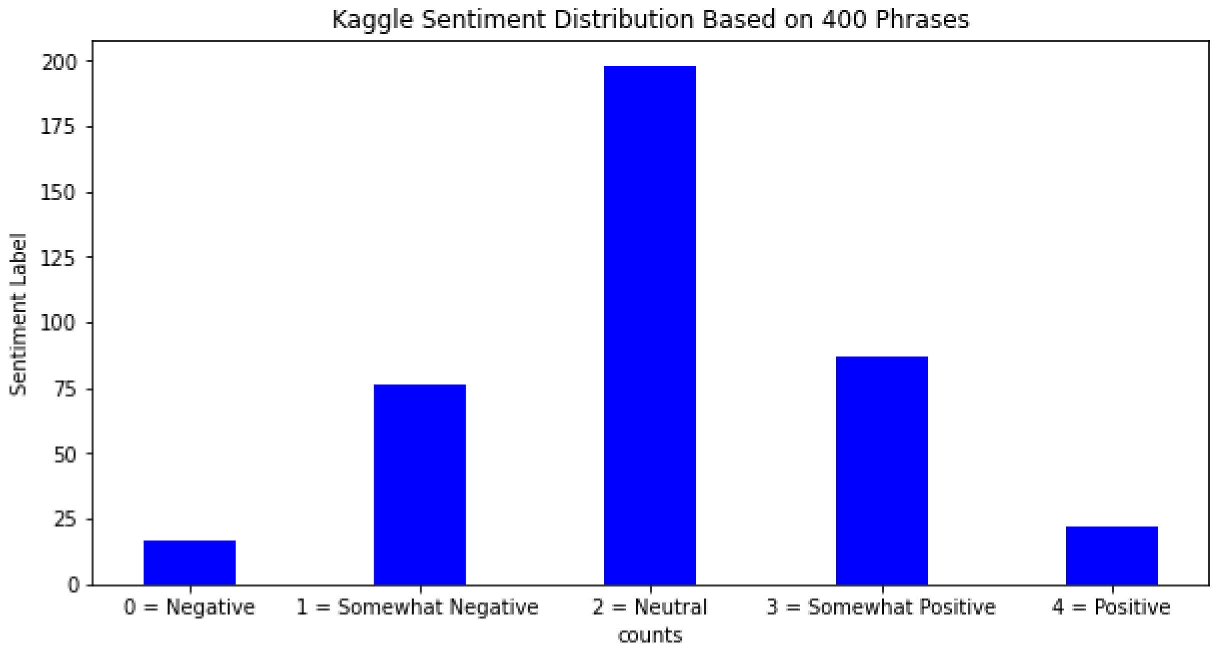


Figure 15: Sentiment Distribution Based on 400 Phrases

b. Featureset/Classifier Results: Precision, Recall, and F1 Per Label

Below is the output from each classifier by the labels as negative = ‘0’, somewhat negative =’1’, neutral =’2’, somewhat positive = ‘3’, and positive =’4’. For the reminding discussion of the labels will be referred in their numeric designation.

1. Original_featureset

	Average Precision	Recall	F1	Per Label
0	0.000	0.000	0.000	
1	0.112	0.205	0.124	
2	0.906	0.549	0.680	
3	0.124	0.242	0.160	
4	0.000	0.000	0.000	

2. Bigrams Featureset

	Average Precision	Recall	F1	Per Label
0	0.000	0.000	0.000	
1	0.112	0.205	0.124	
2	0.906	0.549	0.680	
3	0.124	0.242	0.160	
4	0.000	0.000	0.000	

3. Negated Featureset

	Average Precision	Recall	F1	Per Label
0	0.000	0.000	0.000	
1	0.121	0.205	0.133	
2	0.906	0.551	0.682	
3	0.124	0.242	0.160	
4	0.000	0.000	0.000	

4. POS Featureset

	Average Precision	Recall	F1	Per Label
0	0.000	0.000	0.000	
1	0.108	0.185	0.125	
2	0.795	0.503	0.613	
3	0.161	0.347	0.207	
4	0.000	0.000	0.000	

All classifiers reflected on how the initial data distribution as showed in figure 15. The average precision (Figure 16), recall (Figure 17), and F1 (Figure 18) show no values in sentiment labels, ‘0’ and ‘4’. Only values are showed labels ‘1’, ‘2’, and ‘3’. The label ‘3’, the neutral, has the highest value. Both labels, ‘1’ and ‘3’, have higher recall than precision, indicating that actual True Positive (TP) is answered than predicted. In label ‘2’, precision is higher than recall, showing that actual True Positive (TP) is predicted than answered. F1 value shows that label ‘2’ seemed to have the highest accuracy than the other labels. The results indicated that it is inconclusive to determine which classifiers work the best due to an imbalance of data.

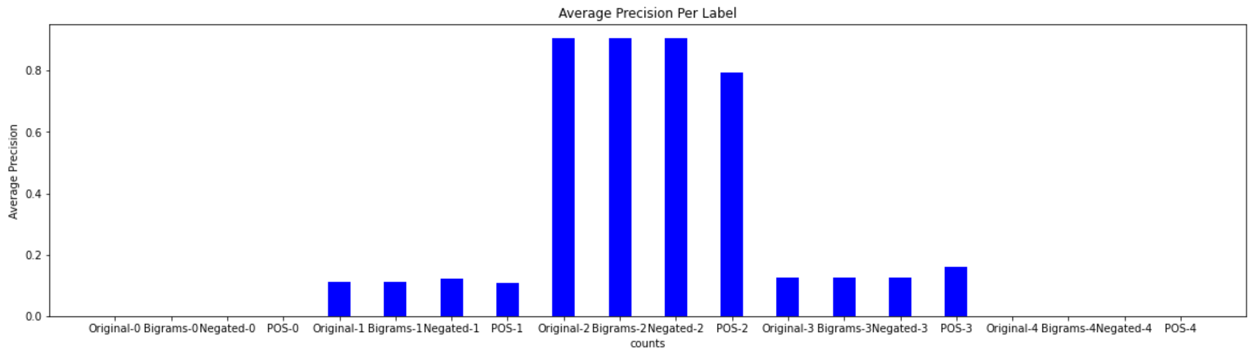


Figure 16: Average Precision By Labels

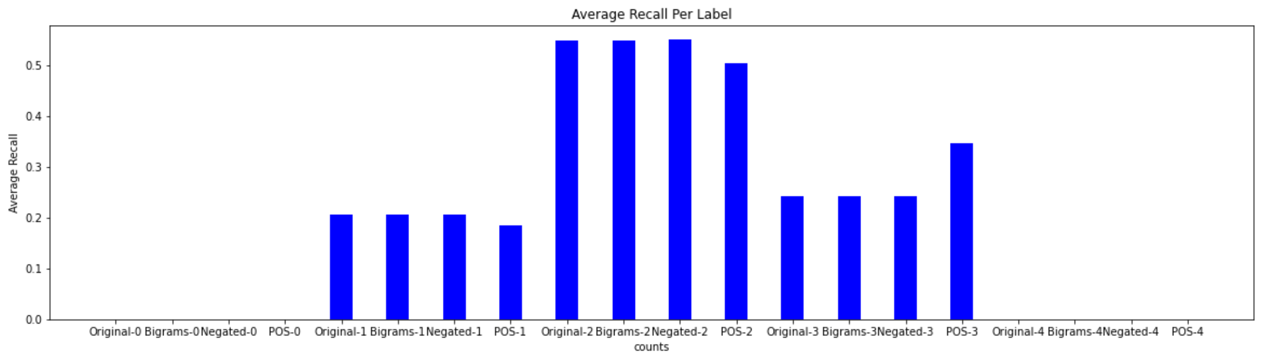


Figure 17: Average Recall By Labels

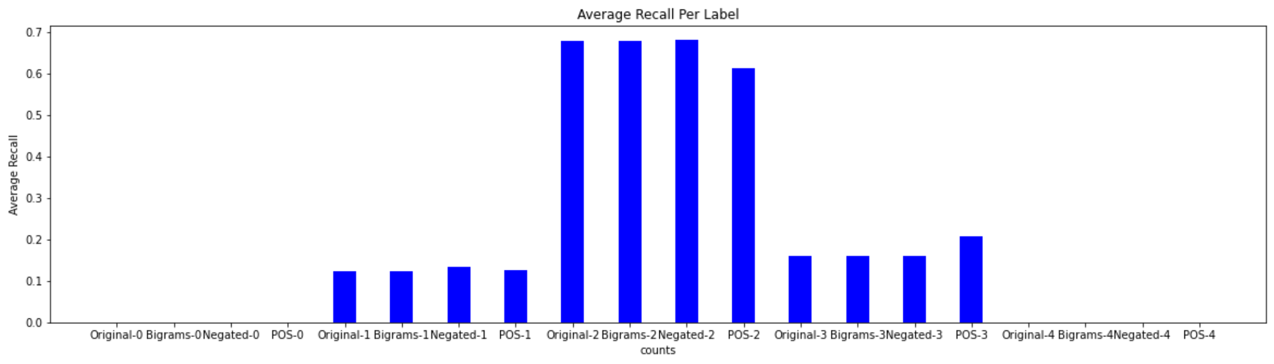


Figure 18: Average F1 By Labels

c. Macro and Micro Average Results

Below is the output from each classifier by Macro and Micro average. The Macro averages are based on the total precision, recall, or F1 divided by the 5 labels. The Micro average incorporates the weight in each label to adjust the value fairly. Then, based on the total weighted precision, recall, or F1 divided by the 5 labels.

1. Original_featureset

Macro Average Precision	Recall	F1	Over All Labels
0.228	0.199	0.193	
Micro Average Precision	Recall	F1	Over All Labels
0.498	0.358	0.394	

2. Bigrams Featureset

Macro Average Precision	Recall	F1	Over All Labels
0.228	0.199	0.193	
Micro Average Precision	Recall	F1	Over All Labels
0.498	0.358	0.394	

3. Negated Featureset

Macro Average Precision	Recall	F1	Over All Labels
0.230	0.199	0.195	

Micro Average Precision	Recall	F1	Over All Labels
0.500	0.359	0.396	

4. POS Featureset

Macro Average Precision	Recall	F1	Over All Labels
0.213	0.207	0.189	

Micro Average Precision	Recall	F1	Over All Labels
0.449	0.352	0.370	

Macro average generated lower precision (Figure 19), recall (Figure 20), and F1 (Figure 21) than the Micro average. The Marco average F1 presented a low accuracy, which indicates poor precision and poor recall. The Micro average with weight-adjusted per label improved the accuracy by ~ 20% in precision, ~ 15% in the recall, and ~ 20% in F1. With the improved Micro average F1 value, the precision is better than recall. However, the weight-adjusted F1 in the Micro average still presents a low accuracy.

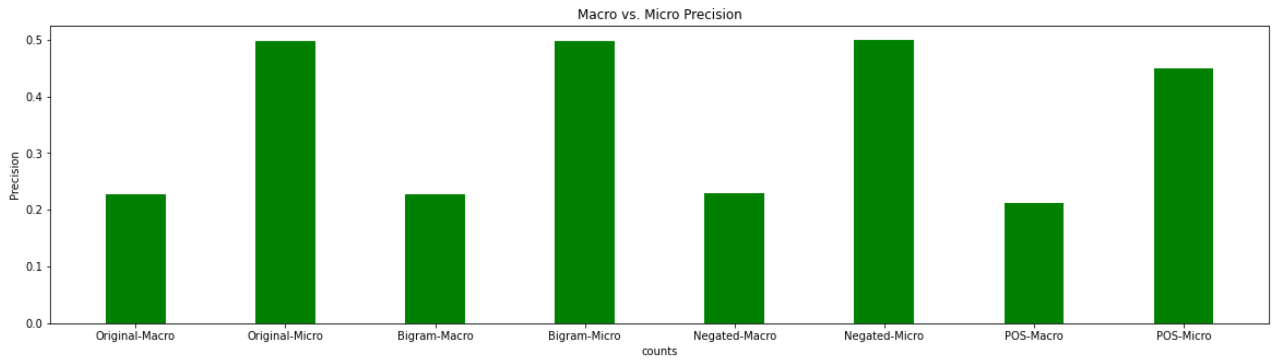


Figure 19: Macro vs. Micro Precision

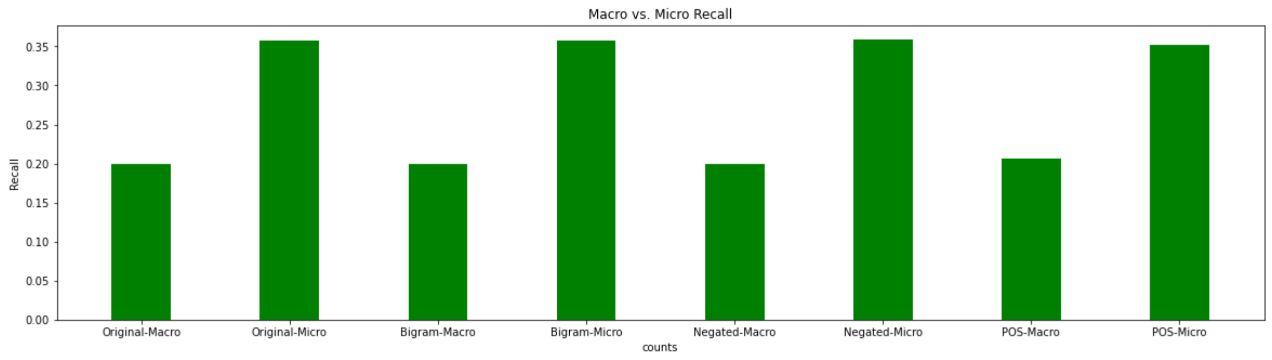


Figure 20: Marco vs. Micro Recall

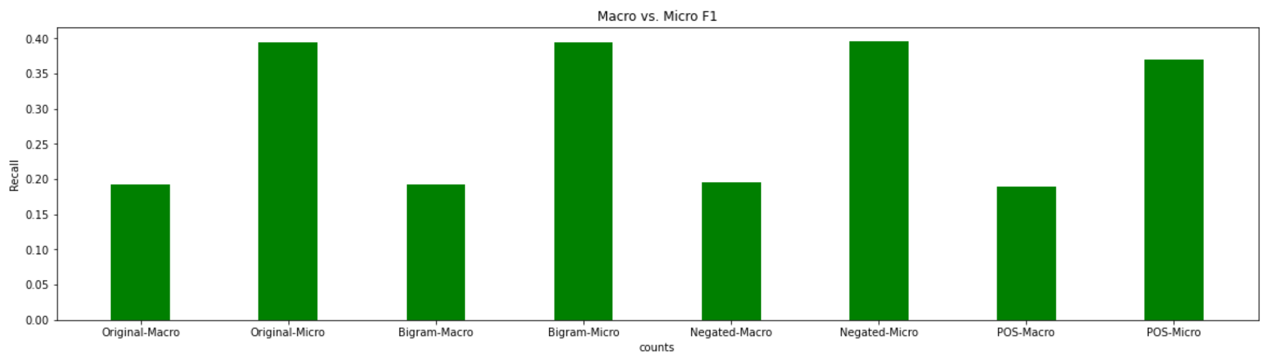


Figure 21: Macro vs. Micro F1

Conclusion:

This final project aims to determine which classifiers/ featuresets are the best for the Kaggle movie review data. Original_featuresets, Bigram_featuresets, Negated_featuresets, and POS_featuresets were evaluated by label had provided low F1 values due to imbalance data. Marco average is further confirmed with an average of low F1 value. Incorporating the weight-adjusted for each label did improve the Micro average F1 value. However, the accuracy is still lacking for all the classifiers.   Therefore, none of the four classifiers might be the best classifier for the Kaggle movie review. Alternative classifiers/ featuresets might need to be further explored with this data for improving the accuracy.