Anya Patel & Matthew Nigro
May 05, 2021
Larche

TEXT CLASSIFICATION USING KAGGLE MOVIE REVIEW PHRASE DATA, LABELED FOR SENTIMENT

1. TEXT PROCESSING

A. Importing packages

i. Though this is not truly a part of text processing, we begin with importing the necessary packages. These include: pandas (as pd), nltk, os, sys, random, and stopwords from nltk.corpus.

B. Reading in the file

i. Here we read in the train.tsv file, which we will so aptly use as it is named—to train our dataset.

ii. We define the vocabulary size by setting the limitStr to 20,000. It was found that under approximately 10,000 was not suitable to work with our algorithm. We will revisit this, however, to discuss how changing the vocabulary size using limitStr effects results of this project.

iii. We now convert the limit argument from a string to an integer.

iv. The train.tsv file is opened and looped over the lines in the file. The first line starting with 'Phrase' is ignored and all lines are read. The sentence IDs are ignored, and the phrase and sentiment are retained. Now, a random sample of the length of the defined limit is chosen, and named phraselist. This is necessary due to overlapping phrase sequences:

```
f = open('/Users/anya/Documents/Grad_local/cis668/final_project/data/kagglemoviereviews/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 the phrase and sentence ids, and keep the phrase and sentiment
      phrasedata.append(line.split('\t')[2:4])

# pick a random sample of length limit because of phrase overlapping sequences
random.shuffle(phrasedata)
phraselist = phrasedata[:limit]

print('Read', len(phrasedata), 'phrases, using', len(phraselist), 'random phrases')
```

C. Viewing phrases

i. We view the phrases from our previously defined phrase list:

```
for phrase in phraselist[:10]:
    print (phrase)
```

```
['A singularly off-putting romantic comedy .', '1']
['at least a decent attempt', '2']
['more and more frustrated and detached as Vincent became more and more abhorrent', '1']
['Personal Velocity', '3']
['hypertime in reverse', '1']
['angst-ridden', '1']
['look American angst', '2']
['But not a whit more .', '2']
['the news of his illness', '2']
["On the surface a silly comedy , Scotland , PA would be forgettable if it were n't such a clever adaptation of the b
ard 's tragic play .", '3']
```

ii. We then create a list of phrase documents as (word, label). This list is named 'phrasedocs'. This list tokenizes the phrases from 'phraselist'.

```
# create list of phrase documents as (list of words, label)
phrasedocs = []
# add all the phrases
for phrase in phraselist:
    tokens = nltk.word_tokenize(phrase[0])
    phrasedocs.append((tokens, int(phrase[1])))

# print a few
for phrase in phrasedocs[:10]:
    print(phrase)
```

```
(['felt', ';', 'not', 'the', 'Craven', 'of', "'", 'A', 'Nightmare', 'on', 'Elm', 'Street', "'", 'or', '`', 'The', 'Hi
lls', 'Have', 'Eyes', ',', "'", 'but', 'the', 'sad', 'schlock', 'merchant', 'of', '`', 'Deadly', 'Friend'], 1)
([',', 'you', "'ll", 'find', 'yourself', 'remembering', 'this', 'refreshing', 'visit', 'to', 'a', 'Sunshine', 'Stat
e', '.'], 3)
(['a', 'frighteningly', 'fascinating', 'contradiction'], 3)
(['with', 'a', 'haunting', 'sense', 'of', 'malaise'], 2)
(['of', 'the', '20th', 'century'], 2)
(['the', 'character', 'he', 'plays'], 2)
(['too', 'many', 'nervous', 'gags'], 1)
(['sometimes', 'wry'], 2)
(['sum', 'up', 'the', 'strange', 'horror', 'of', 'life'], 2)
(['that', "'s", 'rarely', 'as', 'entertaining', 'as', 'it', 'could', 'have', 'been'], 2)
```

2. FEATURE ENGINEERING

A. Using Unigrams

i. We begin our feature engineering by creating a list of unigrams. We get all the words from train.tsv and put them into a frequency distribution. Here we

enforce lowercasing each word, but we do not use stemming or stopwords. The frequency distribution finds the 2000 most frequently appearing keywords in the corpus.

```
all_words_list = [word for (sent,cat) in phrasedocs for word in sent]
all_words = nltk.FreqDist(all_words_list)
# get the 2000 most frequently appearing keywords in the corpus
word_items = all_words.most_common(2000)
word_features = [word for (word,count) in word_items]
print(word_features[:50])
```

ii. We now define features, or keywords, of a document for a BOW/unigram baseline. Each feature consists of 'contains'(keyword) and is true or false depending on whether that keyword is in the document.

```
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
```

iii. Next, we get the feature sets for a document, including keyword features and category features. We also look at the feature sets, but as they are 2000 words long they are just slightly too long to display here.

```
featuresets = [(document_features(d, word_features), c) for (d, c) in phrasedocs]
```

iv. Finally, we enact training using a Naïve Bayes classifier. The training set is approximately 90% of the data.

```
train_set, test_set = featuresets[int((len(featuresets)*0.1)):], featuresets[:int((len(featuresets)*0.1))]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

v. The accuracy of the classifier is now found, as well as the most informative features of the classifier. Note that the accuracy result may vary since the documents are randomized,

```
nltk.classify.accuracy(classifier, test_set)
```

```
0.46
```

```
classifier.show_most_informative_features(30)
```

```
Most Informative Features
                  V_best = True                4 : 2      =     27.8 : 1.0
               V_complex = True                4 : 2      =     21.6 : 1.0
                  V_else = True                0 : 2      =     20.3 : 1.0
                V_should = True                0 : 2      =     20.3 : 1.0
                    V_wo = True                0 : 2      =     20.3 : 1.0
           V_performance = True                4 : 2      =     16.7 : 1.0
                   V_bad = True                0 : 2      =     15.7 : 1.0
             V_adventure = True                4 : 2      =     15.4 : 1.0
               V_perfect = True                4 : 2      =     15.4 : 1.0
          V_performances = True                4 : 2      =     15.4 : 1.0
              V_powerful = True                4 : 2      =     15.4 : 1.0
                 V_quite = True                4 : 2      =     15.4 : 1.0
                  V_seat = True                4 : 2      =     15.4 : 1.0
          V_surprisingly = True                4 : 2      =     15.4 : 1.0
                 V_those = True                4 : 2      =     15.4 : 1.0
                   V_yet = True                4 : 2      =     15.4 : 1.0
                  V_'ll = True                0 : 2      =     14.5 : 1.0
                   V_90 = True                0 : 2      =     14.5 : 1.0
                 V_bore = True                0 : 2      =     14.5 : 1.0
           V_disgusting = True                0 : 2      =     14.5 : 1.0
                 V_dumb = True                0 : 2      =     14.5 : 1.0
            V_horrible = True                0 : 2      =     14.5 : 1.0
            V_material = True                0 : 2      =     14.5 : 1.0
              V_really = True                0 : 2      =     14.5 : 1.0
                 V_set = True                0 : 2      =     14.5 : 1.0
          V_unpleasant = True                0 : 2      =     14.5 : 1.0
            V_watching = True                0 : 2      =     14.5 : 1.0
                 V_very = True                3 : 2      =     13.0 : 1.0
                  V_fun = True                4 : 2      =     13.0 : 1.0
                V_humor = True                0 : 2      =     12.2 : 1.0
```

B. Using Cross Validation to get Precision, Recall, and F-Measure Scores

i. Now we will begin cross-validation. The function 'cross_validation_accuracy' takes the number of folds and the feature sets and iterates over the folds, using different sections for training and testing in turn. At the end, it prints the accuracy for each fold as well as the mean accuracy over all rounds.

```
def cross_validation_accuracy(num_folds, featuresets):
    subset_size = int(len(featuresets)/num_folds)
    print('Each fold size:', subset_size)
    accuracy_list = []
    # iterate over the folds
    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 and save accuracy
        accuracy_this_round = nltk.classify.accuracy(classifier, test_this_round)
        print (i, accuracy_this_round)
        accuracy_list.append(accuracy_this_round)
    # find mean accuracy over all rounds
    print ('mean accuracy', sum(accuracy_list) / num_folds)
```

ii. The next function we define is 'eval_measures_saved'. This function computes precision, recall, and F1 scores for each label (for any number of labels). The input is a list of gold labels, and a list of predicted labels, in corresponding order to one another. The output prints precision, recall, and F1 scores for each label.

```
def eval_measures_saved(gold, predicted):
    # get a list of labels
    labels = list(set(gold))
    # 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
        recall = TP / (TP + FP)
        precision = TP / (TP + FN)
        recall_list.append(recall)
        precision_list.append(precision)
        F1_list.append( 2 * (recall * precision) / (recall + precision))

    return precision_list, recall_list, F1_list
```

iii. Then we define the function 'eval_measures'. This function takes in the parameters of gold and predicted, meaning it has the same inputs as 'eval_measures' saved. While the majority of these two functions are identical, this function, that is 'eval_measures', differs in that instead of a return statement, it prints and formats the precision, recall, and F1 scores into lists. We then create a table of evaluation measures with one row per label.

```
def eval_measures(gold, predicted):
    # get a list of labels
    labels = list(set(gold))
    # 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
        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
    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]))
```

iv. Finally, we find the mean scores across all rounds.

```
labels = list(set(goldlist))

a = precision_scoring[0]
b = precision_scoring[1]
c = precision_scoring[2]
d = precision_scoring[3]
e = precision_scoring[4]
precision_scoring_mean = [(g + h + i + j + k) / 5 for g, h, i, j, k in zip(a, b, c, d, e)]

a = recall_scoring[0]
b = recall_scoring[1]
c = recall_scoring[2]
d = recall_scoring[3]
e = recall_scoring[4]
recall_scoring_mean = [(g + h + i + j + k) / 5 for g, h, i, j, k in zip(a, b, c, d, e)]

a = F1_scoring[0]
b = F1_scoring[1]
c = F1_scoring[2]
d = F1_scoring[3]
e = F1_scoring[4]
F1_scoring_mean = [(g + h + i + j + k) / 5 for g, h, i, j, k in zip(a, b, c, d, e)]
```

v. Displayed below are our relevant outputs, specifically from the 'cross_validation_accuracy' function:

```
Each fold size: 4000
0 0.53925
        Precision       Recall          F1
0             0.186      0.168       0.177
1             0.254      0.401       0.311
2             0.811      0.624       0.705
3             0.293      0.446       0.354
4             0.248      0.353       0.291
1 0.53675
        Precision       Recall          F1
0             0.311      0.196       0.241
1             0.260      0.412       0.319
2             0.804      0.642       0.714
3             0.246      0.384       0.300
4             0.216      0.282       0.244
2 0.54625
        Precision       Recall          F1
0             0.318      0.276       0.296
1             0.232      0.344       0.277
2             0.826      0.637       0.719
3             0.247      0.446       0.318
4             0.251      0.296       0.272
3 0.5565
```

```
        Precision       Recall          F1
0               0.251       0.174       0.206
1               0.253       0.442       0.322
2               0.834       0.644       0.727
3               0.279       0.452       0.345
4               0.315       0.400       0.352
4 0.545
        Precision       Recall          F1
0               0.250       0.261       0.256
1               0.250       0.358       0.295
2               0.811       0.629       0.708
3               0.284       0.461       0.352
4               0.261       0.346       0.298

mean accuracy 0.54475

        Mean Scores
        Precision       Recall          F1
0               0.263       0.215       0.235
1               0.250       0.391       0.305
2               0.817       0.635       0.715
3               0.270       0.438       0.334
4               0.258       0.336       0.292
```

3. EXPERIMENTS

A. Bigrams

i. We now add bigram features. We import needed packages (nltk.collocations) and define the BigramAssocMeasures() function from said package as ‘‘bigram_measures’.

ii. We then create a bigram finder on all the words in sequence. Next, we define the top 500 bigrams using the chi-squared measure.

iii. We now create the ‘bigram_document_features’ function. This function takes the list of words in a document as an argument and returns a feature dictionary. It is dependent upon the variables ‘word_features’ and ‘bigram_features’.

```
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
```

iv. The function is now used to create feature sets for all sentences. Then, it is trained using the Naïve Bayes classifier.

```
train_set, test_set = bigram_featuresets[int((len(featuresets)*0.1)):], bigram_featuresets[:int((len(featuresets)*0.1))
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

v. The metrics consisting of precision, recall, and F1 scores are now found from the previously defined 'cross_validation_accuracy' function. The output is displayed below:

```
Each fold size: 4000
0 0.53925
        Precision       Recall          F1
0             0.186       0.168       0.177
1             0.254       0.401       0.311
2             0.811       0.624       0.705
3             0.293       0.446       0.354
4             0.248       0.353       0.291
1 0.53675
        Precision       Recall          F1
0             0.311       0.196       0.241
1             0.260       0.412       0.319
2             0.804       0.642       0.714
3             0.246       0.384       0.300
4             0.216       0.282       0.244
2 0.546
        Precision       Recall          F1
0             0.318       0.276       0.296
1             0.232       0.344       0.277
2             0.826       0.637       0.719
3             0.247       0.445       0.318
4             0.247       0.292       0.268
3 0.5565
        Precision       Recall          F1
0             0.251       0.174       0.206
1             0.253       0.442       0.322
2             0.834       0.644       0.727
3             0.279       0.452       0.345
4             0.315       0.400       0.352
4 0.545
        Precision       Recall          F1
0             0.250       0.261       0.256
1             0.250       0.358       0.295
2             0.811       0.629       0.708
3             0.284       0.461       0.352
4             0.261       0.346       0.298

mean accuracy 0.5447

        Mean Scores
        Precision       Recall          F1
0             0.263       0.215       0.235
1             0.250       0.391       0.305
2             0.817       0.635       0.715
3             0.270       0.438       0.334
4             0.257       0.335       0.291
```

B. Removing Stopwords

i. The initial stopwords are found using NLTK's build in stopwords. These stopwords are then removed from the list of all words. A 'new_all_words' dictionary is defined, which gets the 2000 most common words as 'new_words_features'.

```
# remove stop words from the all words list
new_all_words_list = [word for (sent,cat) in phrasedocs for word in sent if word not in stopwords]
```

```
# continue to define a new all words dictionary, get the 2000 most common as new_word_features
new_all_words = nltk.FreqDist(new_all_words_list)
new_word_items = new_all_words.most_common(2000)
```

```
new_word_features = [word for (word,count) in new_word_items]
print(new_word_features[:30])
```

```
[',', '.', "'s", 'film', 'movie', 'The', "n't", 'one', '--', '```', 'like', 'A', "'", 'story', '-RRB-', '-LRB-', 'muc
h', 'good', 'characters', '...', 'I', '`', 'time', 'funny', 'comedy', 'way', 'little', 'even', 'movies', 'life']
```

ii. One of the feature set definitions is now re-run with 'new_word_features' instead of 'word_features'.

```
featuresets = [(document_features(d, new_word_features), c) for (d, c) in phrasedocs]
```

iii. Now, utilizing 'feature_sets', we train using the Naïve Bayes classifier.

```
train_set, test_set = featuresets[int((len(featuresets)*0.1)):], featuresets[:int((len(featuresets)*0.1))]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

iv. The metrics consisting of precision, recall, and F1 scores are now found from the previously defined 'cross_validation_accuracy' function. The output is displayed below:

```
Each fold size: 4000
0 0.5465
        Precision       Recall          F1
0               0.122       0.180       0.146
1               0.251       0.411       0.312
2               0.835       0.619       0.711
3               0.293       0.444       0.353
4               0.223       0.331       0.266
1 0.55475
        Precision       Recall          F1
0               0.256       0.236       0.246
1               0.255       0.445       0.324
2               0.833       0.642       0.725
3               0.286       0.403       0.334
4               0.190       0.280       0.226
2 0.5625
        Precision       Recall          F1
0               0.250       0.319       0.280
1               0.242       0.390       0.299
2               0.846       0.635       0.725
3               0.295       0.470       0.363
4               0.199       0.287       0.235
3 0.5645
        Precision       Recall          F1
0               0.184       0.224       0.202
1               0.250       0.433       0.317
2               0.856       0.635       0.729
3               0.289       0.453       0.353
4               0.283       0.400       0.332
4 0.55075
        Precision       Recall          F1
0               0.202       0.304       0.243
1               0.231       0.360       0.281
2               0.827       0.627       0.713
3               0.304       0.457       0.365
4               0.245       0.331       0.282

mean accuracy 0.5558

        Mean Scores
        Precision       Recall          F1
0               0.203       0.253       0.223
1               0.246       0.408       0.307
2               0.839       0.632       0.721
3               0.294       0.445       0.354
4               0.228       0.326       0.268
```

C. Removing Negation Words

i. In order to remove negation words, we first define the list of negation words: 'no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone', 'rather', 'hardly', 'scarcely', 'rarely', 'seldom', 'neither', 'nor'.

ii. To deal with negation words, we utilize the strategy of going through the document words in order and adding the word features. However, if the word follows a negation word, we change the feature (or word) to being classified as a negated word.

```
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
```

iii. Next, feature sets are created using the 'NOT_featuresets' function.

```
NOT_featuresets = [(NOT_features(d, word_features, negationwords), c) for (d, c) in phrasedocs]
# show the values of a couple of example features
print(NOT_featuresets[0][0]['V_NOTcare'])
print(NOT_featuresets[0][0]['V_always'])
```

```
False
False
```

iv. We retrain using the Naïve Bayes classifier.

```
train_set, test_set = NOT_featuresets[int((len(featuresets)*0.1)):], NOT_featuresets[:int((len(featuresets)*0.1))]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

v. Lastly, the metrics consisting of precision, recall, and F1 scores are now found from the previously defined 'cross_validation_accuracy' function. The output is displayed below:

```
Each fold size: 4000
0 0.51725
        Precision       Recall          F1
0             0.324      0.153      0.208
1             0.284      0.402      0.333
2             0.738      0.659      0.696
3             0.274      0.439      0.338
4             0.357      0.273      0.310
1 0.5195
        Precision       Recall          F1
0             0.451      0.168      0.245
1             0.291      0.427      0.346
2             0.734      0.678      0.705
3             0.260      0.397      0.314
4             0.293      0.227      0.256
2 0.52775
        Precision       Recall          F1
0             0.426      0.203      0.275
1             0.262      0.365      0.305
2             0.754      0.671      0.710
3             0.263      0.438      0.329
4             0.342      0.248      0.288
3 0.53175
        Precision       Recall          F1
0             0.397      0.157      0.225
1             0.286      0.420      0.340
2             0.747      0.685      0.715
3             0.285      0.452      0.350
4             0.406      0.310      0.352
4 0.5225
        Precision       Recall          F1
0             0.380      0.192      0.255
1             0.253      0.358      0.297
2             0.733      0.665      0.697
3             0.307      0.464      0.369
4             0.349      0.274      0.307
```

```
mean accuracy 0.52375

        Mean Scores
        Precision       Recall          F1
0             0.396      0.175       0.242
1             0.275      0.394       0.324
2             0.741      0.672       0.705
3             0.278      0.438       0.340
4             0.349      0.267       0.302
```

D. Removing Punctuation

i. We now define a list of 'punctuation_words'. This allows us to remove punctuation marks being taken in as words.

```
punctuation_words = []

for (phrase,score) in phrasedocs:
    for word in phrase:
        if not word.isalpha() and len(word) == 1:
            punctuation_words.append(word)
punctuation_words = list(set(punctuation_words))
```

ii. We redefine the new_all_words_list and remove the punctuation words. We now redefine the 'new_all_words' dictionary with the new list and get the 2000 most common features. We also redefine 'new_word_features' using this. Finally, we redefine 'featuresets' in order to re-run the Naïve Bayes classifier with.

iii. Lastly, the metrics consisting of precision, recall, and F1 scores are now found from the previously defined 'cross_validation_accuracy' function. The output is displayed below:

```
Each fold size: 4000
0 0.5405
        Precision       Recall          F1
0             0.186      0.187       0.187
1             0.268      0.416       0.326
2             0.815      0.619       0.704
3             0.286      0.435       0.345
4             0.214      0.352       0.266
1 0.5415
        Precision       Recall          F1
0             0.299      0.202       0.241
1             0.265      0.416       0.323
2             0.812      0.644       0.718
3             0.256      0.393       0.310
4             0.190      0.262       0.220
2 0.54775
        Precision       Recall          F1
0             0.295      0.283       0.289
1             0.235      0.341       0.278
2             0.826      0.634       0.717
3             0.261      0.455       0.332
4             0.238      0.307       0.268
3 0.558
```

```
        Precision       Recall          F1
0               0.223       0.169       0.193
1               0.264       0.443       0.331
2               0.835       0.643       0.726
3               0.287       0.455       0.352
4               0.295       0.401       0.340
4 0.549
        Precision       Recall          F1
0               0.240       0.273       0.256
1               0.267       0.376       0.312
2               0.814       0.631       0.711
3               0.285       0.458       0.352
4               0.257       0.344       0.295

mean accuracy 0.54735

        Mean Scores
        Precision       Recall          F1
0               0.249       0.223       0.233
1               0.260       0.398       0.314
2               0.820       0.634       0.715
3               0.275       0.439       0.338
4               0.239       0.333       0.278
```

E. POS Tag Features

i. A function is now defined that takes a document list of words and returns a feature dictionary. It runs the default POS tagger (the Stanford tagger) on the document and counts 4 types of pos tags to use as features.

```
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
```

ii. The feature sets are defined using this function to POS_featuresets.

```
POS_featuresets = [(POS_features(d, word_features), c) for (d, c) in phrasedocs]
```

iii. The Naïve Bayes classifier is trained using 'POS_featuresets'.

```
train_set, test_set = POS_featuresets[int((len(POS_featuresets)*0.1)):], POS_featuresets[:int((len(POS_featuresets)*0.1
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

iv. The metrics consisting of precision, recall, and F1 scores are now found from the previously defined 'cross_validation_accuracy' function. The output is displayed

below:

```
Each fold size: 4000
0 0.52725
        Precision       Recall          F1
0               0.250       0.172       0.203
1               0.250       0.389       0.304
2               0.793       0.625       0.699
3               0.264       0.431       0.328
4               0.261       0.323       0.288
1 0.5325
        Precision       Recall          F1
0               0.348       0.183       0.239
1               0.271       0.418       0.329
2               0.790       0.650       0.713
3               0.241       0.390       0.298
4               0.228       0.255       0.241
2 0.5405
        Precision       Recall          F1
0               0.330       0.238       0.276
1               0.253       0.353       0.295
2               0.815       0.644       0.719
3               0.223       0.431       0.294
4               0.277       0.286       0.281
3 0.55
        Precision       Recall          F1
0               0.291       0.158       0.204
1               0.276       0.445       0.341
2               0.814       0.651       0.724
3               0.269       0.455       0.338
4               0.311       0.382       0.343
4 0.538
        Precision       Recall          F1
0               0.288       0.242       0.263
1               0.275       0.371       0.316
2               0.794       0.633       0.705
3               0.256       0.439       0.323
4               0.286       0.335       0.309

mean accuracy 0.53765

        Mean Scores
        Precision       Recall          F1
0               0.301       0.198       0.237
1               0.265       0.395       0.317
2               0.801       0.641       0.712
3               0.251       0.429       0.316
4               0.273       0.316       0.292
```

F. Using a sentiment lexicon with scores or counts: Subjectivity

i. We now proceed to use a sentiment lexicon. We begin by defining a function ‘readSubjectivity’ that returns a dictionary where one can look up words and get back the four items of subjectivity information:

1. *strength, which will be either 'strongsubj' or 'weaksubj'*
2. *posTag, either 'adj', 'verb', 'noun', 'adverb', 'anypos'*
3. *isStemmed, either true or false*
4. *polarity, either 'positive', 'negative', or 'neutral*

```
def readSubjectivity(path):
    flexicon = open(path, 'r')
    # initialize an empty dictionary
    sldict = { }
    for line in flexicon:
        fields = line.split()   # default is to split on whitespace
        # split each field on the '=' and keep the second part as the value
        strength = fields[0].split("=")[1]
        word = fields[2].split("=")[1]
        posTag = fields[3].split("=")[1]
        stemmed = fields[4].split("=")[1]
        polarity = fields[5].split("=")[1]
        if (stemmed == 'y'):
            isStemmed = True
        else:
            isStemmed = False
        # put a dictionary entry with the word as the keyword
        #     and a list of the other values
        sldict[word] = [strength, posTag, isStemmed, polarity]
    return sldict
```

ii. We import the Subjectivity.py program as a module to use the function.

iii. Next, we define features that include word counts of subjectivity words. Negative features have number of weakly negative added to two times the number of strongly negative words. The positive feature has its own corresponding definition.

```
def SL_features(document, word_features, SL):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    # count variables for the 4 classes of subjectivity
    weakPos = 0
    strongPos = 0
    weakNeg = 0
    strongNeg = 0
    for word in document_words:
        if word in SL:
            strength, posTag, isStemmed, polarity = SL[word]
            if strength == 'weaksubj' and polarity == 'positive':
                weakPos += 1
            if strength == 'strongsubj' and polarity == 'positive':
                strongPos += 1
            if strength == 'weaksubj' and polarity == 'negative':
                weakNeg += 1
            if strength == 'strongsubj' and polarity == 'negative':
                strongNeg += 1
            features['positivecount'] = weakPos + (2 * strongPos)
            features['negativecount'] = weakNeg + (2 * strongNeg)
    return features
```

iv. We define the feature sets from the Subjectivity lexicon as ‘SL_featuresets’.

```
SL_featuresets = [(SL_features(d, word_features, SL), c) for (d, c) in phrasedocs]
```

v. We now train using the Naïve Bayes classifier and ‘SL_featuresets’.

```
train_set, test_set = SL_featuresets[int((len(SL_featuresets)*0.1)):], SL_featuresets[:int((len(SL_featuresets)*0.1))]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

vi. The metrics consisting of precision, recall, and F1 scores are now found from the previously defined ‘cross_validation_accuracy’ function. The output is displayed below:

```
Each fold size: 4000
0 0.54925
        Precision       Recall          F1
0               0.234       0.195       0.213
1               0.303       0.421       0.353
2               0.755       0.677       0.714
3               0.407       0.442       0.424
4               0.303       0.305       0.304
1 0.54125
        Precision       Recall          F1
0               0.354       0.209       0.262
1               0.329       0.427       0.372
2               0.729       0.692       0.710
3               0.371       0.392       0.381
4               0.272       0.288       0.279
2 0.56125
        Precision       Recall          F1
0               0.335       0.274       0.302
1               0.314       0.370       0.340
2               0.768       0.686       0.725
3               0.367       0.465       0.410
4               0.329       0.333       0.331
3 0.56025
        Precision       Recall          F1
0               0.313       0.194       0.240
1               0.290       0.411       0.340
2               0.761       0.697       0.728
3               0.417       0.454       0.435
4               0.358       0.386       0.371
4 0.554
        Precision       Recall          F1
0               0.274       0.263       0.268
1               0.334       0.386       0.358
2               0.739       0.681       0.709
3               0.408       0.458       0.432
4               0.332       0.339       0.335


 mean accuracy 0.5532

         Mean Scores
         Precision       Recall          F1
 0               0.302       0.227       0.257
 1               0.314       0.403       0.352
 2               0.750       0.687       0.717
 3               0.394       0.442       0.416
 4               0.319       0.330       0.324
```

G. LIWC Sentiment Lexicon

i. We now use another sentiment lexicon: LIWC. This one returns two lists, words in the positive emotion class and words in the negative emotion class.

```python
def read_words():
  poslist = []
  neglist = []

  flexicon = open('/Users/anya/Documents/Grad_local/cis668/final_project/data/kagglemoviereviews/SentimentLexicons/liwc
  # read all LIWC words from file
  wordlines = [line.strip() for line in flexicon]
  # each line has a word or a stem followed by * and numbers of the word classes it is in
  # word class 126 is positive emotion and 127 is negative emotion
  for line in wordlines:
    if not line == '':
      items = line.split()
      word = items[0]
      classes = items[1:]
      for c in classes:
        if c == '126':
          poslist.append( word )
        if c == '127':
          neglist.append( word )
  return (poslist, neglist)
```

ii. We now create a function to test to see if a word is on a list. This function returns a Boolean true or false value.

```
def isPresent(word, emotionlist):
  isFound = False
  # loop over all elements of list
  for emotionword in emotionlist:
    # test if a word or a stem
    if not emotionword[-1] == '*':
      # it's a word!
      # when a match is found, can quit the loop with True
      if word == emotionword:
        isFound = True
        break
    else:
      # it's a stem!
      # when a match is found, can quit the loop with True
      if word.startswith(emotionword[0:-1]):
        isFound = True
        break
  # end of loop
  return isFound
```

iii. We define features (or words) of a document for a BOW/Unigram baseline. Each feature is 'contains(keyword)' and is true or false depending on whether the keyword is in the document.

```
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
```

iv. We define features that include word counts of subjectivity words. As was used in the Subjectivity Lexicon, negative features have number of weakly negative added to two times the number of strongly negative words. The positive feature has its own corresponding definition.

```
def SL_features_LIWC(document, word_features, pos_features, neg_feautres):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    # count variables for the 4 classes of subjectivity
    Pos = 0
    Neg = 0

    for word in document_words:
        if word in pos_features:
            Pos += 1
            features['positivecount'] = Pos
        elif word in neg_feautres:
            Neg -= 1
            features['negativecount'] = Neg

    return features
```

v. A new feature set is created from this called 'SL_featuresets_LIWC'.

```
SL_featuresets_LIWC = [(SL_features_LIWC(d, word_features, pos_list,neg_list), c) for (d, c) in phrasedocs]
```

vi. We train using the Naïve Bayes classifier and 'SL_featuresets_LIWC'.

```
train_set, test_set = SL_featuresets_LIWC[int((len(
    SL_featuresets_LIWC)*0.1)):], SL_featuresets_LIWC[:int((len(SL_featuresets_LIWC)*0.1))]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

vii. We find the mean accuracy across 5 folds using the 'cross_validation_accuracy' function.

```
Each fold size: 4000
0 0.538
1 0.5375
2 0.549
3 0.554
4 0.545
mean accuracy 0.5447
```

H. Removing Stopwords & Punctuation, then Using Subjectivity Lexicon

i. Here we remove all the stopwords and punctuation that we previously defined from the 'new_all_words_list'. We continue to define a new all words dictionary and et the 2000 most common features as 'new_words_features'.

ii. We now include the Subjectivity Lexicon via 'SL_features' using new words, as is displayed below:

```
featuresets_combined = [(SL_features(d, new_word_features, SL), c) for (d, c) in phrasedocs]
```

iii. The new 'featuresets_combined' and Naïve Bayes classifier are now used for training.

```
train_set, test_set = featuresets_combined[int((len(
    featuresets_combined)*0.1)):], featuresets_combined[:int((len(featuresets_combined)*0.1))]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

iv. We find the mean accuracy across 5 folds using the 'cross_validation_accuracy' function.

```
Each fold size: 4000
0 0.54925
1 0.54575
2 0.563
3 0.564
4 0.5635
mean accuracy 0.5570999999999999
```

I. Removing Stopwords and Punctuation, then Using LIWC Sentiment Lexicon

i. We repeat the process we just went through with the Subjectivity Lexicon. We now include the LIWC Lexicon via 'document_features' using new words, as is displayed below:

```
featuresets_stop_punct = [(document_features(d, new_word_features), c) for (d, c) in phrasedocs]
```

ii. The new 'featuresets_stop_punct' and Naïve Bayes classifier are now used for training.

```
train_set, test_set = featuresets_stop_punct[int((len(
    featuresets_stop_punct)*0.1)):], featuresets_stop_punct[:int((len(featuresets_stop_punct)*0.1))]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

iii. We find the mean accuracy across 5 folds using the 'cross_validation_accuracy' function.

```
Each fold size: 4000
0 0.54675
1 0.5565
2 0.558
3 0.57275
4 0.55875
mean accuracy 0.55855
```

4. RESULTS & OUTCOME

A. Cross-Validation Accuracy Analysis

i. Via analysis of the 5-fold cross-validation mean results we found that the following steps had the most significant impact on increasing accuracy. Without adding on any methods, we received an initial mean accuracy on our Naïve Bayes classifier of 0.54475. From this, we could find the net positive impact on accuracy as we introduced each method.

1. *Creating bigrams and utilizing these in our classification allowed for a mean accuracy of 0.5447. This was actually lower than the initial baseline accuracy by -0.0005.*
2. *Removing stopwords allowed for a mean accuracy of 0.5558. This when compared to the baseline accuracy allowed for a net increase in accuracy of 0.0111.*
3. *Removing negation words allowed for a mean accuracy of 0.52375. This when compared to the baseline accuracy allowed for a net decrease in accuracy of -0.02095.*
4. *Removing punctuation words allowed for a mean accuracy of 0.54735. This when compared to the baseline accuracy allowed for a net decrease in accuracy of -0.02095.*
5. *Utilizing POS Tag Features allowed for a mean accuracy of 0.53765. This when compared to the baseline accuracy allowed for a net decrease in accuracy of -0.00711.*
6. *Using the Subjectivity Lexicon allowed for a mean accuracy of 0.5532. This when compared to the baseline accuracy allowed for a net increase in accuracy of 0.00845.*

7. *Using the LIWC Sentiment Lexicon allowed for a mean accuracy of 0.5447. This when compared to the baseline accuracy allowed for a net decrease in accuracy of -0.0005.*
8. *When we removed stopwords and punctuation, then used the Subjectivity lexicon we found an accuracy of 0.5570999999999999. This when compared to the baseline accuracy allowed for a net increase in accuracy of 0.01235.*
9. *When we removed stopwords and punctuation, then used the LIWC Sentiment lexicon we found an accuracy 0.55855. This when compared to the baseline accuracy allowed for a net increase in accuracy of 0.0138.*

ii. This demonstrated that combining removal of stopwords with removal of punctuation in congruence with the LIWC Sentiment Lexicon was most effective in increasing accuracy, followed closely by the combined removal of stopwords and removal of punctuation in congruence with the Subjectivity Lexicon. As an isolated experiment, removing stop words was most effective in increasing accuracy.

B. Precision, Recall, and F1-Scores Analysis

i. Via analysis of the 5-fold cross-validation mean results we found that the following steps had the most significant impact on precision. Without adding on any methods, we received a final mean precision on our Naïve Bayes classifier of 0.258.

1. *Creating bigrams and utilizing these in our classification allowed for a final mean precision of 0.257.*
2. *Removing stopwords allowed for a final mean precision of 0.228.*
3. *Removing negation words allowed for a final mean precision of 0.349.*
4. *Removing punctuation allowed for a final mean precision of 0.239*
5. *Utilizing POS Tag Features allowed for a final mean precision of 0.237*
6. *Using the Subjectivity Lexicon allowed for a final mean precision of 0.319.*
    a. *It is important to note that the initial precision of our Naïve Bayes model had a low score. This means very many false positives to begin with, as it is only 0.258. However, removing negation words was most successful in limiting false positives and raised precision to 0.349. Additionally, using the Subjectivity lexicon was also successful in limiting false positives and raised precision to 0.319. However, removing stopwords lowered the precision the most, then did using POS Tag Features, and then removing punctuation. Creating bigrams only marginally decreased precision by 0.1, so it could be said to have little effect.*

ii. Via analysis of the 5-fold cross-validation mean results we found that the following steps had the most significant impact on recall. Without adding on any methods, we received a final mean recall on our Naïve Bayes classifier of 0.336.

1. *Creating bigrams and utilizing these in our classification allowed for a final mean recall of 0.335.*
2. *Removing stopwords allowed for a final mean recall of 0.268.*
3. *Removing negation words allowed for a final mean recall of 0.302.*
4. *Removing punctuation allowed for a final mean recall of 0.278.*
5. *Utilizing POS Tag Features allowed for a final mean recall of 0.316.*
6. *Using the Subjectivity Lexicon allowed for a final mean recall of 0.330.*
    a. *It is important to note that the initial recall of our Naïve Bayes model had a low recall. This means very many false negatives to begin with, as it is only 0.336. However, introducing stopwords, though it increased the accuracy, did allow for an even greater amount of false negatives. Removing punctuation also decreased recall significantly. In fact, each method did this, however, creating and using bigrams was the least with only a 0.001 difference from the baseline recall.*

iii. Via analysis of the 5-fold cross-validation mean results we found that the following steps had the most significant impact on F1-scores. Without adding on any methods, we received a final mean F1-scores on our Naïve Bayes classifier of 0.292.

1. *Creating bigrams and utilizing these in our classification allowed for a final mean F1-score of 0.291.*
2. *Removing stopwords allowed for a final mean F1-score of 0.268.*
3. *Removing negation words allowed for a final mean F1-score of 0.302.*
4. *Removing punctuation allowed for a final mean F1-score of 0.278*
5. *Utilizing POS Tag Features allowed for a final mean F1-score of 0.292.*
6. *Using the Subjectivity Lexicon allowed for a final mean F1-score of 0.324.*
    a. *It is important to note that since the initial F1-score is already quite low, at 0.292, and the F1 scores really only marginally improve or worsen, we seem to be missing a great deal of data we would want to analyze.*

C. Changing the Vocabulary Size

i. It was found that when the vocabulary size decreased it marginally decreased the accuracy measure. However, it must be noted that this may be due to chance as it is so very small. The dataset was so large regardless that keeping the vocabulary size at over 10,000 (as this is the low limit at which our algorithm could function), may have had little true impact.

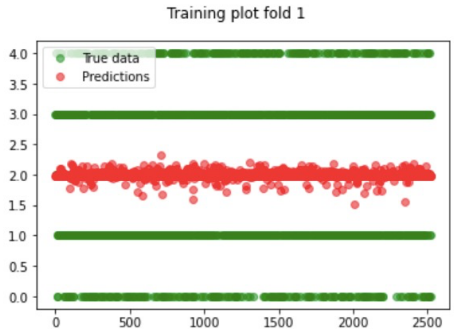
5. ONE LAST EXPERIMENT: AN AI ATTEMPT

A. Overview

i. In this section, we attempted to use numpy, sklearn, pytorch, pandas, and matplotlib in congruence with NLTK in order to perform sentiment analysis. We wanted to create a neural network that would classify words as either strongly positive, weakly positive, strongly negative, or weakly negative. We selected out all movie reviews that were 13 characters in length—if they were more or less than 13 their sentiment would not be classified. Furthermore, this neural network had 13 hidden layers and a learning rate of 0.1. The sentiment would be classified by sorting the sentences (after vectorization) into 4 bins and assigning the numerical values of 0, 1, 2, or 3.

B. Results & Outcome

i. In summation, this attempt did not properly classify as we hoped. When we passed our data through the neural network that we created the output can be graphically represented as shown in the image below:



Essentially, the model gravitated towards the mean, and did not produce viable classification. However, this was still a good experiment and it yielded interesting results (if not accurate results) and significantly furthered our curiousity.

6. OBSERVATIONS & LESSONS LEARNED

A. We learned from the project in its entirety how to use NLTK to perform Sentiment analysis on Kaggle movie review phrase data. We learned how the vocabulary size effected our models and we learned how to experiment with models to get the most optimal results.

B. In the results, we learned the value of analyzing all the different parts of the metrics, rather than just the accuracy (so the precision, recall, and F1-scores) To do this, we had to learn what they really meant—not just a definition, but what they meant in regards to our own project.

C. In our Neural networking attempt, we found that the main issue in our results came from the way we vectorized our movie reviews. The way it was vectorized was too elementary. In order to go further with this process and achieve better results a better vectorizing method would have to be used. We look forward to expanding our studies to hopefully one day be able to revisit these topics with more knowledge under our belt.

D. We would like to note that we did complete the entirety of this project together, working side by side. We did not split it up, but instead collaborated the entire way through on each and every part.