Before getting started with reproducing the steps within the report, please make sure that all the files in the code are set to the appropriate directory. This program is created within jupyter notebook and incompatible with the command prompt usage. Please use jupyter notebook to reproduce the results.

NOTE: The tsv files used in this report are stored under "C:/Users/jiebi/Desktop/Grad/Summer 2021/IST 664/Final/corpus/" for me, but it would be different for everyone else. Please remember to alter this.

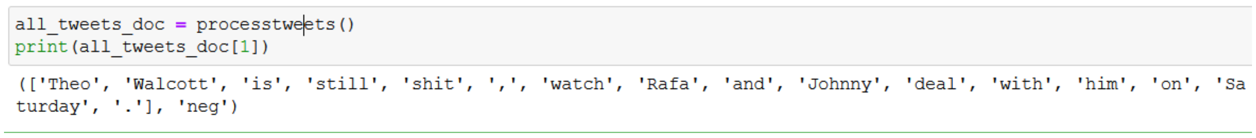
Data Preprocessing

To start off with analyzing the SemEval dataset, a function called "processtweets" was written that does many things for processing this file. First, the raw file *"downloaded-tweeti-b-dist.tsv"* was opened up and first had its "@" and "#" characters removed from its text portion. As this is a twitter dataset it is very common to see those two non-alphanumeric characters in it. Removing this will be very helpful in the next steps. Another thing to note here is that, this function is operating under the assumption that the tweets were already randomized within the original file so that no further randomization is needed later on. This also helps with reproducing the results found in this report.

Next, tweets that were marked as "Not Available" were ignored and removed in this next pre process step. This step involves the usage of the tweet tokenizer. It can be inferred from its name that this is a special word tokenizer used for tweets. That's why this was chosen. Now that the tweets have been turned into word tokens

Since this dataset contained some annotated labels of "positive", "negative", "neutral", "objective", and "objective-or-neutral" it was simplified down to "pos", "neg", and "neu". Neu included "neutral", "objective", and "objective-or-neutral". After obtaining this list of labels along with its appropriate tweets (in word tokens) these two were combined into a single list called "tweetdocs" within the function and was what this returned.

This function was then ran and stored into a new variable. The following screenshot displays an example of what some outputs of this list looks like now.



```
all_tweets_doc = processtweets()
print(all_tweets_doc[1])
```

```
(['Theo', 'Walcott', 'is', 'still', 'shit', ',', 'watch', 'Rafa', 'and', 'Johnny', 'deal', 'with', 'him', 'on', 'Sa
turday', '.'], 'neg')
```

After obtaining the main source of data here this is where the following few portions would differ a bit as it would relate to creating several feature sets to run the functions.

Additional Function Creations

After creating the main source of data that's being analyzed, several functions were created to help create feature sets and obtaining the precision, recall, and F1 scores after running cross validations.

Function to obtain document features:

```python
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
```

This is a very familiar function as it was used in the previous report. What it does is that it takes the input of the "all_tweets_doc" list we had created earlier with all the sentences and also the word features list as well. Then a set of empty dictionaries will be created to capture the Boolean value of whether those words from each sentence would exist within list of the word features. After looping through all the words for each sentence, the features dictionary would be completed here.

<u>Function to calculate precision, recall, and F1:</u>

```
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)
```

The function takes the inputs of 3 different lists that consists of actual results and predicted results of the model and the labels. The lists should only contain tags of "neg", "pos", or "neu" as the indexes would match up. It then groups up the entire list of actual tags to see how many unique labels

there are, which in this case is only 3 with (pos, neg, and neu). Then blank lists were created to capture the precision, recall, and F1 score to it. At the end there would be 3 values in each list here.
Next, the for loops would run 3 times (because there are 3 different tags) and within each for loop the true positives, true negatives, false positives, and false negatives were counted if the actual values match the predicted values. After counting and summing all those values, it will then be entered into mathematical formulas that would capture the recall, precision, and f1 scores. Once that value gets calculated it gets added into the appropriate list.

These outcomes in the list here are also useful in obtaining the micro and macro averages later on in the main function that runs the cross validations.

Function for Cross Validation:

This function has a lot to break down so for simplicity in explanation it will be broken into different chunks. Keep in mind that every piece of code underneath is all in one function (unless otherwise stated).

```
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
```

First the function takes 3 inputs. The inputs are the number of folds that the cross validations should run, the feature sets to run the CV on, and unique list of labels (pos, neg, neu).

It first tries to split up the feature sets by dividing the length of that set with the number of folds that's specified. Afterwards, it then generates an integer number for the number of label that list has and storing it in a new variable. Then 3 lists were created with default values of 3 zeroes (because of the 3 unique labels) to store the precision, recall, and F1 later on.

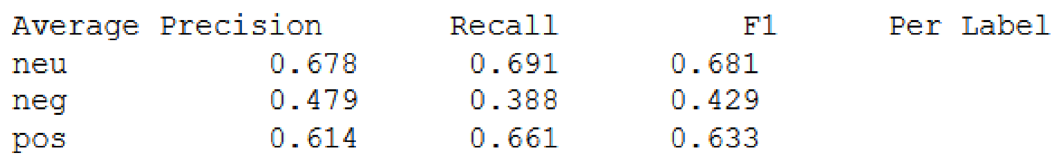
```
for i in range(num_folds):
        test_this_round = featuresets[(i*subset_size):][:subset_size]
        train_this_round = featuresets[:(i*subset_size)] + featuresets[((i+1)*sub
set_size):]
        # train using train_this_round
        classifier = nltk.NaiveBayesClassifier.train(train_this_round)
        # evaluate against test_this_round to produce the gold and predicted labe
ls
        goldlist = []
        predictedlist = []
        for (features, label) in test_this_round:
            goldlist.append(label)
            predictedlist.append(classifier.classify(features))
        (precision_list, recall_list, F1_list) \
                  = eval_measures(goldlist, predictedlist, labels)

        # 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]
```

In the second part of this function, the precision, recall, and F1 would actually get calculated. Within the main part of this loop, the feature sets would get split up into training and test set for each number of fold with the subset size obtained earlier. Once the split occurred, then the Naïve bayes classifier would fire up and start training with the training subset here. After that is complete, 2 blank lists would be created to capture the results for the actual & predicted list later. Within the subloops now, the test_this_round would be classified with the trained classifier and also call the eval_measures function to calculate the precision, recall, and F1 scores for this model. Those 3 lists would be extracted from that function and saved onto the variables within this function. After exiting out of this subloop, the function takes the code into another set of for loops and start changing the default values of 0s in the previous total_precision_list, total_recall_list, and total_F1_list into the appropriate value calculated from the function.

This entire process would be looped for X number of folds, specified by the user. For the purpose of this report, let's make that number of folds = to 5 for a quicker run time since the ancient computer I have takes a very long time to complete a single fold. If you have a stronger machine, please do try setting the folds to 8 for a higher validity of the result.

```
    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]
    # the evaluation measures in a table with one row per label
    print('\nAverage Precision\tRecall\t\tF1 \tPer Label')
    # 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]))
```

This part of the code takes the precision, recall, and F1 obtained earlier and calculate then display the average of it per label. Here's a sample output of what this would look like:



```
Average Precision       Recall          F1      Per Label
neu              0.678    0.691      0.681
neg              0.479    0.388      0.429
pos              0.614    0.661      0.633
```

The next part of this is obtaining the macro average. Which is just an average of all the precision, recall, and F1 values while treating each label equally.

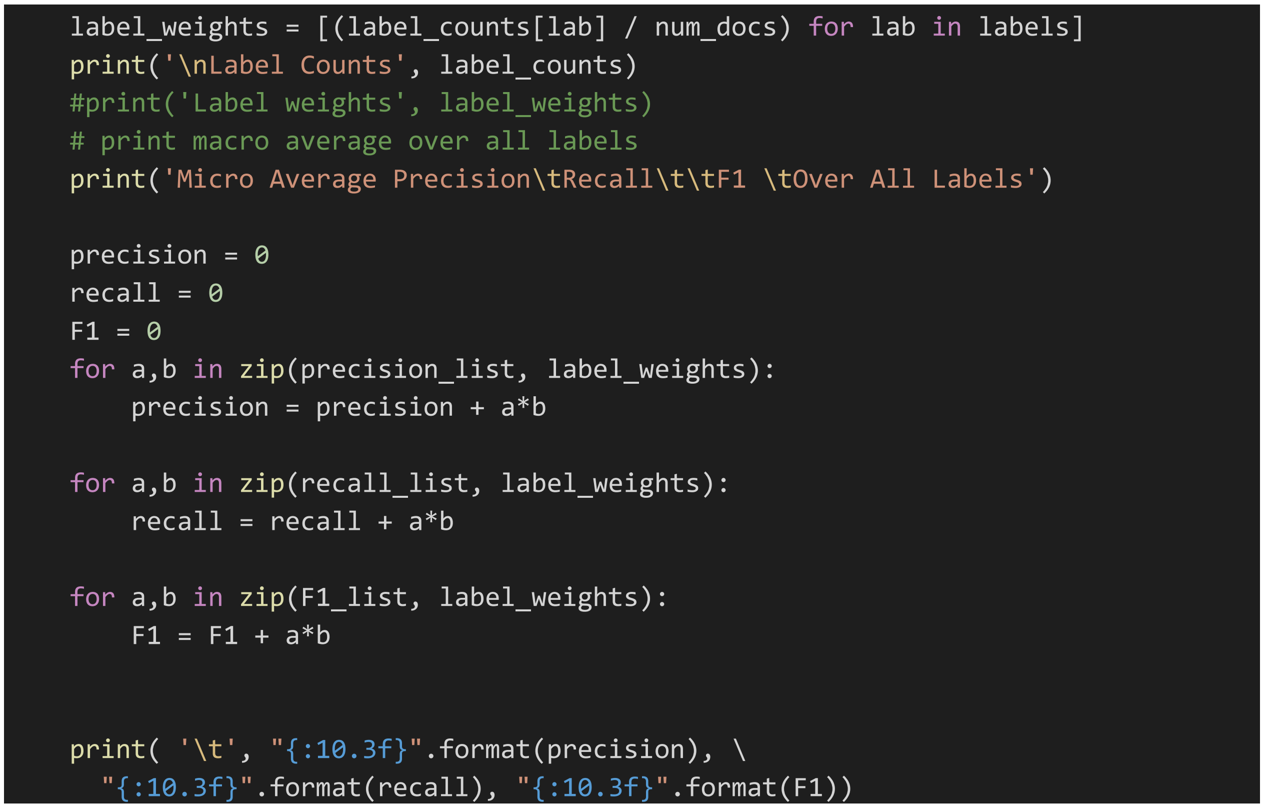
```
print('\nMacro Average Precision\tRecall\t\tF1 \tOver All Labels')

    p_dummy = 0
    r_dummy = 0
    f_dummy = 0

    for x in precision_list:
        p_dummy = p_dummy + x

    for x in recall_list:
        r_dummy = r_dummy + x

    for x in F1_list:
        f_dummy = f_dummy + x

    print('\t', "{:10.3f}".format(p_dummy/num_labels), \
          "{:10.3f}".format(r_dummy/num_labels), \
          "{:10.3f}".format(f_dummy/num_labels))
```

The main part in this is that, dummy variables of 0 need to be created for summing up the values in the lists to calculate the average. The "sum" function didn't work here because those values are float numbers. To work around that issue, this method was created to sum up all the float values per list and divided by the number of labels it had. In this case it is 3. Then these were printed out in a nicer format. Here's a sample display for this portion of the function:

```
Macro Average Precision Recall          F1      Over All Labels
                 0.590       0.580       0.581
```

Lastly, the function also calculates the micro average for precision, recall, and F1. This would help mitigate the effect of skewed data sets.

```
    label_counts = {}
    for lab in labels:
      label_counts[lab] = 0
    # count the labels
    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 = 0
    recall = 0
    F1 = 0
    for a,b in zip(precision_list, label_weights):
        precision = precision + a*b

    for a,b in zip(recall_list, label_weights):
        recall = recall + a*b

    for a,b in zip(F1_list, label_weights):
        F1 = F1 + a*b


    print( '\t', "{:10.3f}".format(precision), \
      "{:10.3f}".format(recall), "{:10.3f}".format(F1))
```

In this part of the function, it counts the number of labels the feature set has and stores it for calculation. The weights of each label are calculated afterwards to ensure each label gets its appropriate weight before averaging them together. Afterwards, the number of each labels are printed out (in the example below, this is an indeed skewed data set with ~1200 neg). To calculate the 3 scores here with the weighted labels, a similar method like calculating the macro average was used since the sum function doesn't work for float numbers. After obtaining the values in precision, recall, and F1 they were printed out nicely. Here's how a sample output looks like:

```
Label Counts {'neu': 3942, 'neg': 1207, 'pos': 3059}
Micro Average Precision Recall          F1      Over All Labels
                 0.625      0.635      0.626
```

This marks the end of the cross_validation_PRF function.

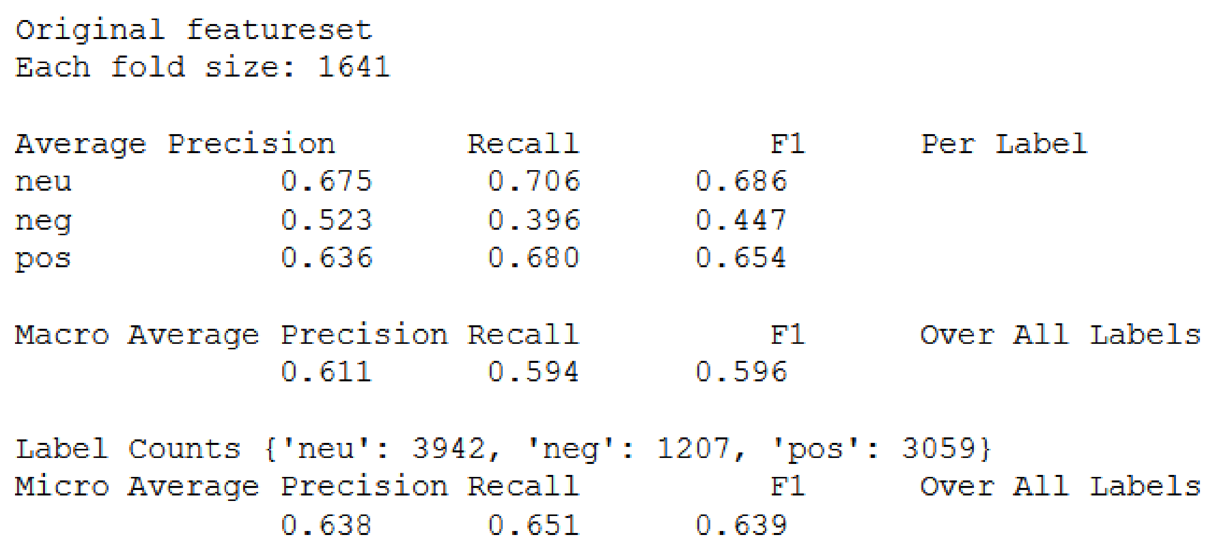
<u>Features Sets</u>

With the main functions being written out, now it's time to start creating feature sets. In these feature sets, the most common 2000 words would be used.

Bag-of-words:

To start off, the bag-of-words unigram feature set is created (a very familiar process now). This would be the baseline measure as well. It's done by setting up the most common 2000 words as the "bag of words" for the features of determining a tweet's label.

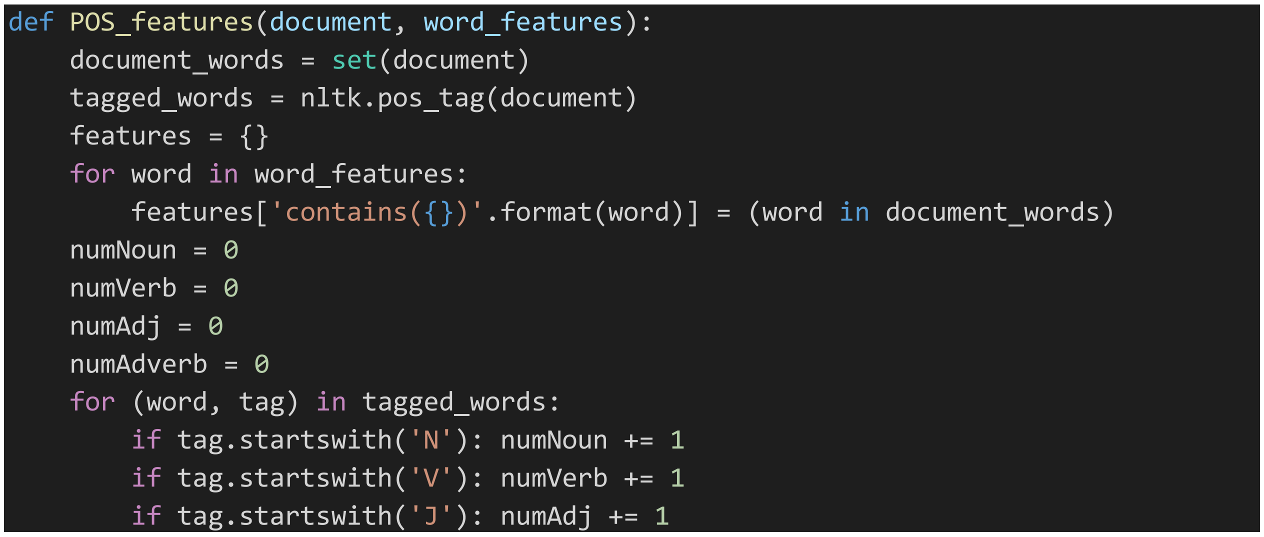
Here are the results of this method:



```
Original featureset
Each fold size: 1641

Average Precision       Recall          F1      Per Label
neu             0.675     0.706     0.686
neg             0.523     0.396     0.447
pos             0.636     0.680     0.654

Macro Average Precision Recall          F1      Over All Labels
                0.611     0.594     0.596

Label Counts {'neu': 3942, 'neg': 1207, 'pos': 3059}
Micro Average Precision Recall          F1      Over All Labels
                0.638     0.651     0.639
```

It's notable that there is skewness in the dataset with a lower amount of neg labels. Without this information it may seem that the neg predictions are just wrong off the charts, but the micro average says otherwise. As a result, when comparing the different feature sets the micro average will be sole indicator here. The other numbers are there for visibility purposes. Generally all three numbers of precision, recall, and F1 are very close within each other. The main number here should be F1 as it balances both the concerns of precision and recall into one. With precision and recall being so close to each other, F1 looks like a good metric for measuring how effective the model was at predicting the "positives" results not to be confused with the "pos" label.

POS:



```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
```

This takes in the input of a document list and also the word features list created from the most common 2000 words. In this, the words within the document tokens would get tagged with POS tags. It then gets stored within a dictionary and goes through list of word features to start tagging it with a Boolean value as the feature set. Afterwards, the count of nouns, verbs, adjectives, and adverbs were added to the dictionary of features and the dictionary would be returned. This function was then ran to create the POSfeaturesets and the feature set was used to run the cross validations on obtaining the precision, recall, and F1. Here are the results:

```
POS featureset
Each fold size: 1641

Average Precision       Recall          F1      Per Label
neu             0.669      0.707      0.683
neg             0.541      0.379      0.442
pos             0.615      0.679      0.643

Macro Average Precision Recall          F1      Over All Labels
                0.608      0.589      0.589

Label Counts {'neu': 3942, 'neg': 1207, 'pos': 3059}
Micro Average Precision Recall          F1      Over All Labels
                0.630      0.649      0.633
```

The results of this feature set is highly similar to the bag-of-words feature sets. The F1 here is just slightly lower than the F1 in bag-of-words by .006. Technically the bag-of-words feature sets is a slightly better feature set to use for classifying the sentiments for these tweets. However, both can be used interchangeably since their difference seem pretty minor here.

Another Experiment Feature Set

It looks like without much preprocessing, the model was still doing an alright job of classification here. Since the POS feature sets were overtaken a bit by the bag-of-words feature sets, some modifications would be made to the original document to see if this would help increase the F1 score here.

The two main changes here would be converting the text to lowercase and also removing any stop words and then creating the new POS feature set to see if it would be a better model. The reasons that these changes are being taken are that people don't take capitalization seriously in social media and the stop words could also be generating extra noise. That's why I believe that applying these two changes would be beneficial to increase the F1 score for the new feature set.

Since the original document features were a tuple, it was impossible to turn it into lowercase without manipulating the original function. However, doing that would harm the reproducibility of this experiment.

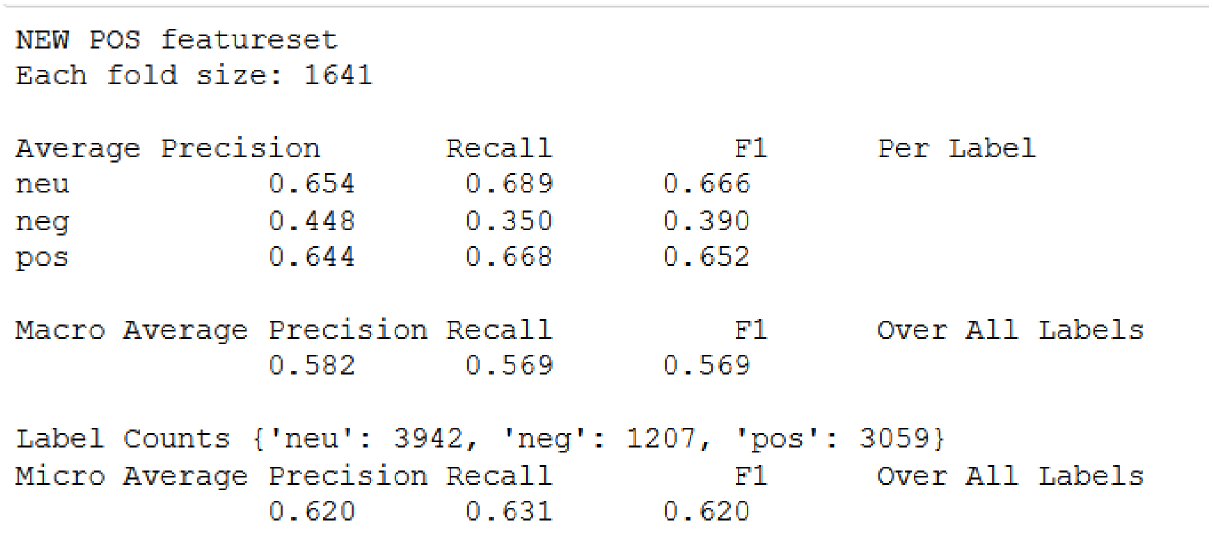
```python
new_tweets_doc = []
for x,y in all_tweets_doc:
    new_tweets_doc.append([x,y])

for i,x in new_tweets_doc:
    for tokens in range(len(i)):
        i[tokens] = i[tokens].lower()
stopwords = nltk.corpus.stopwords.words('english')
stopwords.extend([line.strip() for line in open('C:/Users/jiebi/Desktop/Grad/Summer 2021/IST 664/Final/stopwords_twitter.txt')])
newstopwords = [word for word in stopwords]

new_all_words_list = [word for (sent,cat) in new_tweets_doc for word in sent if word not in newstopwords]
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]
new_POSfeaturesets = [(document_features(d, new_word_features), c) for (d, c) in all_tweets_doc]
```

With those concerns in mind, the original document was copied over and stored as a list of lists rather than a list of tuples to apply the .lower() function. After turning every text into lowercase, then the stopword lists from NLTK was applied in addition to the twitter stop list.

NOTE: To reproduce this, please change the path to where the stopwords_twitter.txt file exists in your computer.

Then it was just a matter of obtaining a new word list and ultimately the new_POSfeaturesets here.

Below is the new precision, recall, and F1 from the “new_POSfeaturesets”:

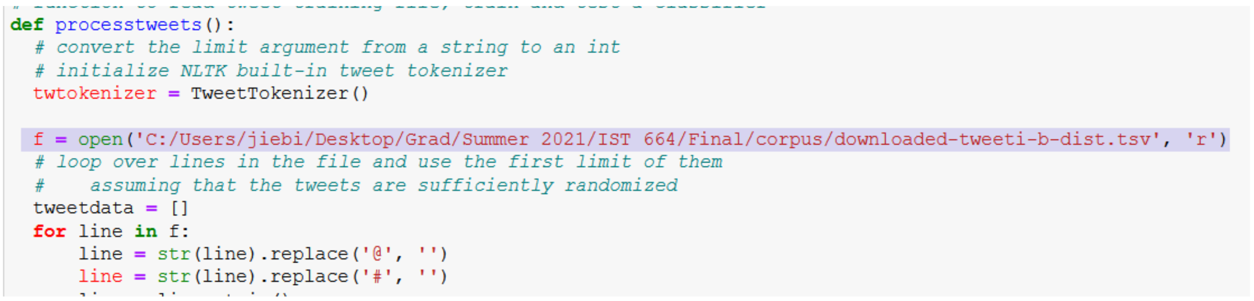


```
NEW POS featureset
Each fold size: 1641

Average Precision       Recall          F1      Per Label
neu             0.654       0.689       0.666
neg             0.448       0.350       0.390
pos             0.644       0.668       0.652

Macro Average Precision Recall          F1      Over All Labels
                0.582       0.569       0.569

Label Counts {'neu': 3942, 'neg': 1207, 'pos': 3059}
Micro Average Precision Recall          F1      Over All Labels
                0.620       0.631       0.620
```

From this new method, the changes weren’t that drastic again. However, it did went down with the additional manipulation of removing stop words and changing the cases in the tweets. The result that can be inferred from this specific example is that making those drastic changes won’t affect this model as much and would make it slightly worse.

<u>Another Dataset</u>

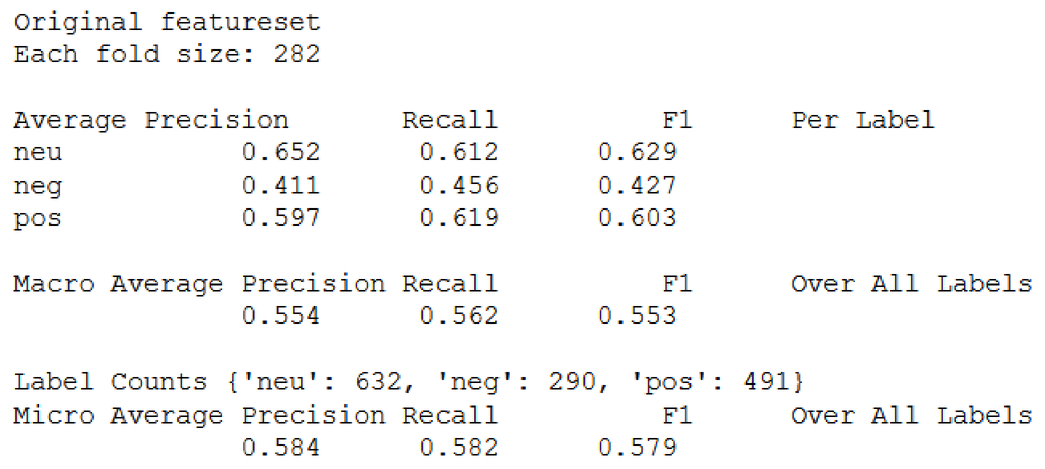
Let’s do this all again with another data set. To do this, the path of the file in processtweets function was modified a little bit from 'C:/Users/jiebi/Desktop/Grad/Summer 2021/IST 664/Final/corpus/downloaded-tweeti-b-dist.tsv' → 'C:/Users/jiebi/Desktop/Grad/Summer 2021/IST 664/Final/corpus/downloaded-tweeti-b.dev.dist.tsv'



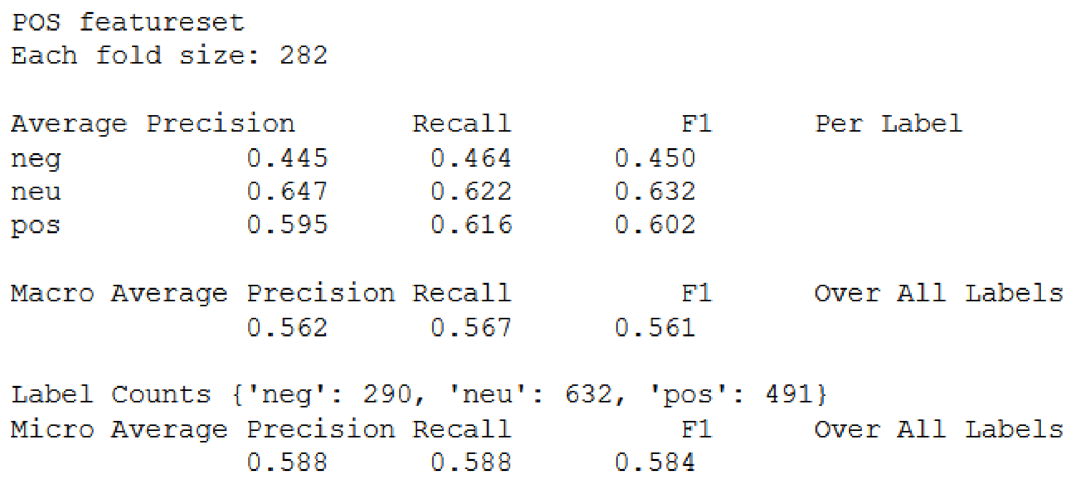
```
def processtweets():
  # convert the limit argument from a string to an int
  # initialize NLTK built-in tweet tokenizer
  twtokenizer = TweetTokenizer()

  f = open('C:/Users/jiebi/Desktop/Grad/Summer 2021/IST 664/Final/corpus/downloaded-tweeti-b-dist.tsv', 'r')
  # loop over lines in the file and use the first limit of them
  #    assuming that the tweets are sufficiently randomized
  tweetdata = []
  for line in f:
      line = str(line).replace('@', '')
      line = str(line).replace('#', '')
```

If circumstances would allow it, I would train the model with a full set of tweets from the b-dist.tsv file and test it on b-dist-dev.tsv file to obtain better results. The limit of completing this is due to my ancient computer’s processing power and doing that might have the possibility of overheating and crashing (it has happened once already). To prevent that from happening, the best I was able to do is to switch up the datasets and run all 3 experiments again to see how big of a difference it makes on the F1 when the dataset changes. With a lower number of tweets in the dev file here, the hypothesis here is that F1 would be significantly lower by 20% or so. This hypothesis is created because there might be insufficient data to train the model, and causing inaccuracies.

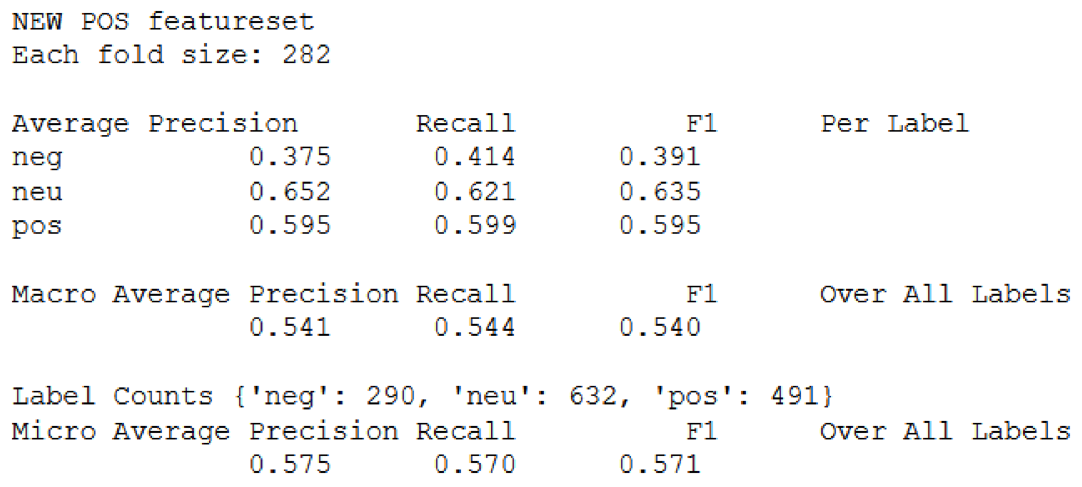
<u>Results – Another experiment</u>



```
Original featureset
Each fold size: 282

Average Precision       Recall          F1      Per Label
neu             0.652      0.612      0.629
neg             0.411      0.456      0.427
pos             0.597      0.619      0.603

Macro Average Precision Recall          F1      Over All Labels
                0.554      0.562      0.553

Label Counts {'neu': 632, 'neg': 290, 'pos': 491}
Micro Average Precision Recall          F1      Over All Labels
                0.584      0.582      0.579
```



```
POS featureset
Each fold size: 282

Average Precision       Recall          F1      Per Label
neg             0.445      0.464      0.450
neu             0.647      0.622      0.632
pos             0.595      0.616      0.602

Macro Average Precision Recall          F1      Over All Labels
                0.562      0.567      0.561

Label Counts {'neg': 290, 'neu': 632, 'pos': 491}
Micro Average Precision Recall          F1      Over All Labels
                0.588      0.588      0.584
```



```
NEW POS featureset
Each fold size: 282

Average Precision       Recall          F1      Per Label
neg             0.375      0.414      0.391
neu             0.652      0.621      0.635
pos             0.595      0.599      0.595

Macro Average Precision Recall          F1      Over All Labels
                0.541      0.544      0.540

Label Counts {'neg': 290, 'neu': 632, 'pos': 491}
Micro Average Precision Recall          F1      Over All Labels
                0.575      0.570      0.571
```

From the results of processing everything here on this new experiment it seems that the hypothesis was correct. All the feature sets with the dev.tsv file did produce a lower F1 score than the original dataset that was used. However, one interesting thing to point out here is that, in this new dataset the POS_featuresets actually had the highest F1 score. Its precision and recall scores were identical here. From the results here the conclusion of having a different amount of data in data sets do make a bigger difference than just procuring a different feature set.

Conclusion

This sentiment analysis experiment was the worst nightmare of people with ancient laptops that are 5 years old. This is by far the most challenging problem that I've experienced here.

Other than that issue with computer hardware, it seems like the amount of data in data sets really do alter the F1 scores of the models. From the experiments it seems the different feature sets used in the models don't seem to have that big of an effect here. Another thing to point out is that the precision and recall scores for all experiments were actually pretty close to each other, but it's just easier comparing models using F1 scores. It doesn't make a huge difference here.

Below are some thoughts for making some improvements in the future:

- Obtain a computer with enough processing power that won't overheat and shutdown
- Obtain larger datasets to work with or explore into other categories of datasets
- Create more advanced word feature lists and apply further filters to exclude additional noise
- Running the models with a higher number of folds

This was quite a journey from learning the basics of NLP to creating intense models that are computer breaking (literally) for classification tasks!

Thank you for reading this super long report!