**Dataset used:**

Kaggle movie review: a dataset with manually annotated sentence subphrases ranging from negative to positive.

**Preprocessing:**

Preprocessing began with defining the feature functions. Among those made were document\_features, NOT\_features, and POS\_features.

*Document\_features*

The document\_features function takes the arguments word\_features and document. The document is then passed through the set method create document\_words. The set function passes through the text and returns a dictionary of the unique words that are used. Duplicated words are not passed on to the document\_words variable. The next step in the function is to create an empty dictionary called features. This dictionary will take words from the document\_words variable formatted by the following loop:

A picture containing chart

Description automatically generated

For every word in the document\_words, there will be a feature that shows ‘V\_{word from document\_words}. The ‘.format(word)’ part of the co inputs every word form documents into {} following ‘V\_’. The result of the function is the dictionary of features. The purpose of the function is to see if corpus unigrams appear in any sentence.

*NOT\_features*

The NOT\_features have 3 arguments, two of which, are the same as the document\_features function except for negationwords. Negatiowords is a manually curated list of words that can change the sentiment of a sentence from positive to negative or vice versa. The features dictionary that is created is then divided between ‘V\_{}’ and ‘V\_NOT{}’ with the default value being False. ‘V\_NOT{}’ features will store document words that match the words in the negationwords list or ends in “n’t” while ‘V\_{}’ will store the remaining words through the subsequent for loop. The subsequent feature dictionary is then returned once the for-loop finishes.

Text

Description automatically generated

*POS\_features*

The POS\_features function looks at the part of speech tags for each word in the document argument. Like document\_features function, the POS function applies the set method on the document to get a list of all the words used. The document part of speech tags is created using nltk.pos\_tag and then put into the tagged\_words variable. The tagged\_words variable contains the word and its POS tag. An empty features dictionary is then created to house the speech tag totals that will be calculated with the for loop. The for loop starts by taking the words in document\_words and adds them to features as part of the string, ‘contains({})’. Then four new variables meant to count the number of occurrences for a particular part of speech with 0 as their value. This allows for the second for look to add 1 to the corresponding variable when the matching matches the first letter in the tag.startswith() method. The resulting counts are then coalesced into the features dictionary which is subsequently returned at the end of the function.

**Cross validation:**

The cross-validation function that takes the number of folds, the feature previously made, and the labels created in the function and calculates the performance of each fold, along with the average performance. The variable, subset\_size, the length of the featuresets are divided by the number of fold. This variable be used to create the training and test rounds later in the function. A printed display of the fold size is used to check how it relates to the subsequent performance. the next variable, num\_labels, is created by taking the length of the label argument multiplying it by zero to create the baseline totals for the precision, recall, and F1 lists.

The first for-loop iterates over the folds to make the training and test sets. The test\_this\_round variable is a subset of the featuresets variable with a portion beginning at the product of the fold number and subset\_size and another portion equal to the subset size minus one. The train\_this\_round variable is a combination of two featuresets subsets. The first subset contains the product of the fold number and subset\_size and the second subset contains the featuresets starting at the product of subset\_size and the fold number plus one. The train\_this\_round is then put into the naïve bayes classifier.

Two blank lists are then created, goldlist, to create the gold standard labels, and predictedlist for predicted features. For each feature and label in test\_this\_round the goldlist adds the label and the prediction list adds the classifier’s result give the features as the input. Following that, the function computes the evaluations measures for each fold and returns a list of measures for each label.

Text

Description automatically generated

The precision, recall, and F1 labels are printed next so the next for-loop can add measures to each label. For each label in labels, the measures for each label are printed underneath them. The labels are made into titles using ‘\t’. the second for-loop adds the sums of the three measures to their respective lists.

Text, letter

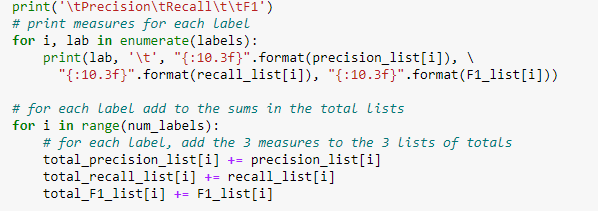
Description automatically generated

For all rounds of cross-validation the totals from the previous label lists are averages by the number of folds and attributed to a list the corresponds to the label.

A picture containing text, indoor

Description automatically generated

Those lists are then made into a table like the one for the measure totals for each round. The titles are the labels with the averages for each round list underneath using the same for-loop as the totals.



The macro average table follows the same syntax as the previous two tables. However, instead of simply formatting the measures into the string the for-loop divides the sum of the lists by the number of labels. This method treats each label equally.

Text

Description automatically generated

Lastly, micro averages, weigh the scores for each label by the number of items which removes some of the imbalances for larger itemsets. An empty dictionary called label\_counts is created and the lab in label\_counts is made equal to zero. For each document and label in featuresets, the label\_counts go up by 1. A variable, num\_docs, holds the length of the featuresets variable to use in the weighting of the micro averages. The label\_weights variable is the quotient of the label counts and the num\_docs variable in the contexts of the label for-loop. To keep track of the label counts for reference a printed tally is used after calculating weights.

Text

Description automatically generated

The resulting table is formatted the same as the previous tables with the calculated output being the product of label\_weights and the three lists. The zip() method takes the lists and the label\_weights and makes one object that will be returned as an output once the function is complete. The efficiency of the micro average will increase as the limit size increases as there is a greater chance of large disparities between rounds.

Text, letter

Description automatically generated

**Eval\_measures function:**

The eval function takes gold, predicted, and general labels, in the same order, as arguments. The function then takes those arguments and returns lists of computed precision, recall and F1 values. Recall, Precision, and F1 are all made into empty lists. For each lab in labels, all equaled zero. For each instance[i] and value(val) in the enumerated gold labels, a series of if statements are run. If both value and predicted [i] = lab then TP goes up by one. If the value equals the lab but, the predicted does not, then FN goes up by one. If the value does not equal lab but the predicted [i] does, then FP goes up by one. If the value or predicted [i] equal the lab then TN goes up by one. These if statements are repeated until the entire gold list is read.

**Text

Description automatically generated**

As a safeguard against dividing by zero, an if statement was written so any label equaled zero would zero out all of the lists. The else statement, for non-zero numbers, calculates the recall, precision, and F1 and appends them to their respective lists. The returned values are the lists in table form with label as row.

**Process Kaggle function:**

The process Kaggle function takes the direct path to the training set and a string limit and runs it through all the previous functions, as well as training and testing the classifier. The limitStr is an integer that limits the number of phrases that will be read in for debugging and testing purposes. Smaller limits <1000 are ideal for debugging while bigger limits >8000 are for data analysis. The dirPath uses the os module to get the current working directory (os.getcwd()) which is then extended to ‘\corpus’. This links the current working directory to the training data within the movie review corpus. The print statement shows the pathway used to access that file. The f variable opens the training file for the for-loop. Before initiating the for-loop a blank list for phrasedata is created. For each line in the corpus, the end of line character is stripped along with the sentence and phrase ids. The first line was ignored with a not line.startwith(‘Phrase’) statement. The data that was appended to the blank phrasedata list were the phrase and sentiment from each line. Due to the overlapping sequence length random.shuffle() was used to establish the length of the variable, phraselist. The print statement shows the length of phrases compared to the length of random phrases from the phraselist variable.

A picture containing diagram

Description automatically generated

The phrasedocs list was then created to hold a list of phrase documents containing lists of words and labels. While there are 4 sentiment labels in the corpus only 3 bins are used for better performance. Using the phrases in the phraselist variable, the tokenized words from the phases are put into the tokens variable. The tokens the sentiment labels are added in list format to the phrasedocs list. The empty list, docs is used to hold the lowercase tokens from the phrasedocs list. The first ten instances are printed to make sure the method worked correctly.

Each word from the sentences in docs are put into an all\_words\_list. These words are then put into the nltk frequency distribution module(nltk.FreqDist()) to create all\_words variable. The length of that variable is printed to make sure no data has been lost. From all\_words variable, the 1500 most common words were put into a variable called word\_items. The word\_features list separates the word from each (word, count) pair in word\_items.

Stopwords are created using the nltk English stopword corpus and the negationwords are manually curated. These two lists were then used to make a newstopwords list containing stopwords that aren’t also part of the negationwords list. The new\_all\_words\_list then removes any words that match those in the newstopwords list. That list is then put through the frequency distribution module to then parse out the 1500 most common words into new\_word\_items. The new\_word\_features list then separates the words from the (word, count) combination.

To prepare for the bigram features function, the features must be created. A finder is used first to find all bigram collocations from all\_words\_list. The top 500 bigrams determined by the chi squared measure are then put into the bigram\_features variable.

**Bigram\_document\_features function:**

The bigram\_document\_features function first uses the set method to make the document\_words variable. Document\_bigrams are created using the nltk.bigrams method on the document argument. An empty features dictionary was created to take in words from document\_words and bigrams from document\_bigrams distinguished using ‘V\_{}’ and ‘B\_{}\_{}’ respectively. The returned features dictionary should have a dictionary for unigrams and bigrams.

**Creating feature sets from the functions:**

Four feature sets were made from the four functions. Each function used d from (d,c) in docs as the document argument and the results were then put into featuresets variables with the prefix matching the function it uses.

Text, letter

Description automatically generated

To prepare for cross validation of the featuresets, a label list was created using the labels from docs as well as a num\_folds variable set to 10.

**Cross-validation of featuresets:**

For each cross-validation, there is a printed string denoting which featureset is being fed into the cross\_validation\_PRF function. In each iteration of the function, the num\_folds variable, the featureset, and the label list were used as arguments. The output was 10 folds with four labels as rows and the list names: precision, recall, and F1 as the column headers.

**Results:**

The results of the code are extracted from the Kaggle subdirectory train.tsv file with a limit of 10,000 random phrases. The first ten random phrases printed as a check shows a relatively even sentiment spread. Sentiment labels 1, 2, and 3 each hade three phrases while there was only one with a 0 rating. Each of the 10 folds used 1/10th or 1000 phrases. For each feature tested, the sentiment rating 2 had the highest cross-validation scores, all scores were above .600 among all labels. All other sentiment ratings ranged between .1 and .5. The overall shape of the cross-validation across folds and features resembles a normal distribution where most of the sentiment is at 2, tapering off at the more extreme ends. This trend matches sentiment label counts for each featureset analysis.

The macro averages were identical across all labels in the original and bigram featuresets. Precision and F1 matched at .351 and recall matched .375. The POS featureset had a similar macro averages to the original and bigram featuresets with precision and F1 average at .351 and recall at .371. Negation had the highest macro averages with .391 for precision, .371 for recall, and .364 for F1.

Like the macro averages, micro averages were identical between unigram and bigram featuresets. Precision, recall, and F1 were 0.529, 0.495, and 0.498 respectively. The POS featureset had minute +0.002 to +0.004 difference from the aforementioned featuresets. The negation featureset had the most variation with a -0.017-difference compared to the bigram and unigram featuresets and +0.005 difference in the F1 category.

**Conclusions:**

Based on the results, the sentiment label 2 had the highest precision, recall, and consequently F measure, the harmonic means of precision and recall. This could be a result of most reviews taking a neutral stance on a movie because it neither impressed nor disgusted in their language. Negation showed the highest of all other feature sets because the negation words most likely made ratings more ambiguous to the machine learning model. With those words removed, words more indicative of overall sentiment were used and thus, raised the accuracy.

The lack of differences between macro and micro averages for the unigram and bigram feature sets show that these bag of word features are not as effective in classifying sentiment without stop words and particularly, negationwords being removed.

For future analysis more emphasis would put on fine tuning the negation words list to improve precision and recall scores.