**Final Project Report**

**Processing and Classification of Sentiment or other Data**

**Dataset-**Kaggle Movie Review

The data was taken from the original Pang and Lee movie review corpus based on reviews from the Rotten Tomatoes web site.

We are going to use training data- “train.tsv”, and test data- “test.tsv”.

train.tsv contains the phrases and their associated sentiment labels. The sentiment labels are:

0 - negative

1 - slightly negative

2 - neutral

3 - slightly positive

4 – positive

test.tsv contains just phrases.

**Goal** of this project- To predict the sentiments of reviews using basic classification algorithms and compare the results by varying different parameters.

**Steps for Classification and Sentiments Analysis-**

1. **Fetch data from train.tsv**

# function to read kaggle training file, train and test a classifier

def processkaggle(dirPath,limitStr):

# convert the limit argument from a string to an int

limit = int(limitStr)

os.chdir(dirPath)

f = open('./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])

1. **Randomize data and select a certain no. of phrases from phrase data.**

# 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')

for phrase in phraselist[:10]:

print (phrase)

Note- Commandline interface takes a directory name with kaggle subdirectory for train.tsv and a limit to the number of kaggle phrases to use

if \_\_name\_\_ == '\_\_main\_\_':

if (len(sys.argv) != 3):

print ('usage: classifyKaggle.py <corpus-dir> <limit>')

sys.exit(0)

processkaggle(sys.argv[1], sys.argv[2])

1. **Tokenization-**

I tried three different types of tokenizers-

• Countvectorizer

from sklearn.feature\_extraction.text import CountVectorizer

def countvector\_token(phraselist):

phrasedocs4 = []

for phrase in phraselist:

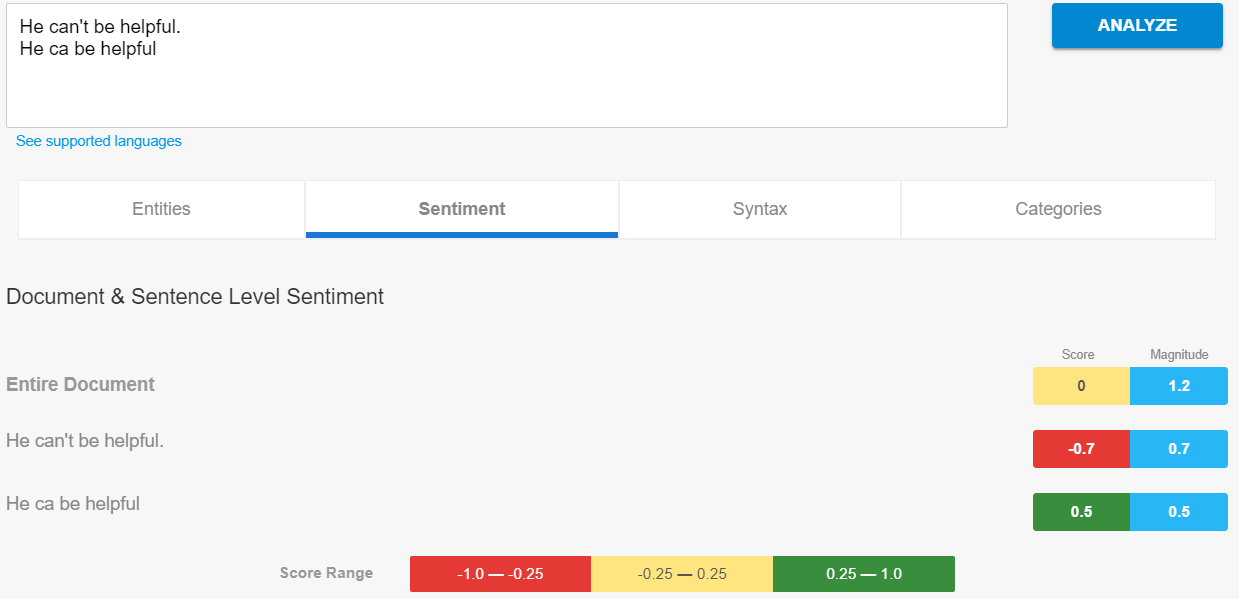
cvtokens = CountVectorizer().build\_tokenizer()(phrase[0])

phrasedocs4.append((cvtokens, int(phrase[1])))

return phrasedocs4

This divided sentences into words similar to wordpunct tokenizer and removed all single character words like ‘ca n’t’ resulted into ‘ca’ and ‘U.S.A.’ was completed removed.

Observation-Change of negative words like can’t to ca won’t be useful in sentiment analysis. For example (Based on Google NL API)-



• word\_punct tokenizer

def wordpunct\_token(phraselist):

phrasedocs3 = []

for phrase in phraselist:

wptokens = nltk.wordpunct\_tokenize(phrase[0])

phrasedocs3.append((wptokens, int(phrase[1])))

return phrasedocs3

This divided word like ‘U.S.A.’ into six words- ‘U’,’.’,’S’,’.’,’A’,’.’ and ‘ca n’t’ into four words- ‘ca’,’n’, ’’’ ,’t’.

• word tokenizer

def word\_token(phraselist):

phrasedocs = []

for phrase in phraselist:

tokens = nltk.word\_tokenize(phrase[0])

phrasedocs.append((tokens, int(phrase[1])))

return phrasedocs

This approach preserved unique words like U.S.A but negative words like ‘woud n’t’ are divided into two parts ‘would’ and ‘n’t’. It would be interesting to see how various classification algorithms react to this

Clearly if we want to do classification without pre-processing then word tokenizer would be the most useful compared to other two.

1. **Pre-processing**

To remove unnecessary words like non-alphanumeric words and stop list, I did some pre-processing on words generated by above tokenizers. I have divided tokenization into two categories-

1. Word tokenizer without pre-processing (as discussed earlier)
2. Word tokenizer with pre-processing

In first one, I have not applied pre-processing on sentences and then created pairs of (tokensOf(sentence), label) list for classification. In second one I have used some pre-processing steps before classification.

**Lower case:**

I converted all tokens into lower case as many functions of nltk are case-sensitive

def lower\_case(doc):

return [w.lower( ) for w in doc]

**Clean text:**

Removing punctuation, stop words and single character would again result in change of negative words like can’t to can. As we have seen in count vector case, this won’t be useful in our sentiment analysis. In order to avoid such scenario, we will need to expand some stop words with apostrophe.

def clean\_text(doc):

cleantext = []

for review\_text in doc:

review\_text = re.sub(r"it 's", "it is", review\_text)

review\_text = re.sub(r"that 's", "that is", review\_text)

review\_text = re.sub(r"\'s", "\'s", review\_text)

review\_text = re.sub(r"\'ve", "have", review\_text)

review\_text = re.sub(r"wo n't", "will not", review\_text)

review\_text = re.sub(r"do n't", "do not", review\_text)

review\_text = re.sub(r"ca n't", "can not", review\_text)

review\_text = re.sub(r"sha n't", "shall not", review\_text)

review\_text = re.sub(r"n\'t", "not", review\_text)

review\_text = re.sub(r"\'re", "are", review\_text)

review\_text = re.sub(r"\'d", "would", review\_text)

review\_text = re.sub(r"\'ll", "will", review\_text)

cleantext.append(review\_text)

return cleantext

**Removing punctuation and numbers-**

As punctuation and numbers will be unnecessary for sentiment analysis

def rem\_no\_punct(doc):

remtext = []

for text in doc:

punctuation = re.compile(r'[-\_.?!/\%@,":;\'{}<>~`\()|0-9]')

word = punctuation.sub("", text)

remtext.append(word)

return remtext

**Removing some stop words-**

I am not going to remove negative words like not, cannot, would not as they will be useful in our sentiment analysis.

from nltk.corpus import stopwords

def rem\_stopword(doc):

stopwords = nltk.corpus.stopwords.words('english')

updatestopwords = [word for word in stopwords if word not in ['not', 'no', 'can','has','have','had','must','shan','do', 'should','was','were','won','are','cannot','does','ain', 'could', 'did', 'is', 'might', 'need', 'would']]

return [w for w in doc if not w in updatestopwords]

**Stemming and Lemmatization-**

In assignment 1, I had tried three stemmers-Lancaster, Porter and Snowball stemmer. I had also examined document using WordNet lemmatizer. I found that Lancaster stemmer was severe on some words like event and ever resulted into ev whereas Snowball stemmer hardly changed any word compared to other two stemmers. The WordNet lemmatizer only removes affixes if the resulting word is in its dictionary like lying remains same instead of changing to lie. So, I decided to use combination of WordNet lemmatization and Porter stemming.

def lemmatizer(doc):

wnl = nltk.WordNetLemmatizer()

lemma = [wnl.lemmatize(t) for t in doc]

return lemma

def stemmer(doc):

porter = nltk.PorterStemmer()

stem = [porter.stem(t) for t in doc]

return stem

So, our final word tokenizer with preprocessing will look like this-

def process\_token(phraselist):

phrasedocs2 = []

for phrase in phraselist:

tokens = nltk.word\_tokenize(phrase[0])

tokens = lower\_case(tokens)

tokens = clean\_text(tokens)

tokens = rem\_no\_punct(tokens)

tokens = rem\_stopword(tokens)

tokens = stemmer(tokens)

tokens = lemmatizer(tokens)

phrasedocs2.append((tokens, int(phrase[1])))

return phrasedocs2

1. **Filtering-**

**Removing 1 and 2 characters-**

Single characters and double character words that might be generated through above mentioned pre-processing won’t be useful in our classification and sentiment analysis.

def rem\_character(doc):

word\_list=[]

for word in doc:

if (len(word) > 1):

word\_list.append(word)

return word\_list

Similarly, for unprocessed tokens we can extract words in following ways:

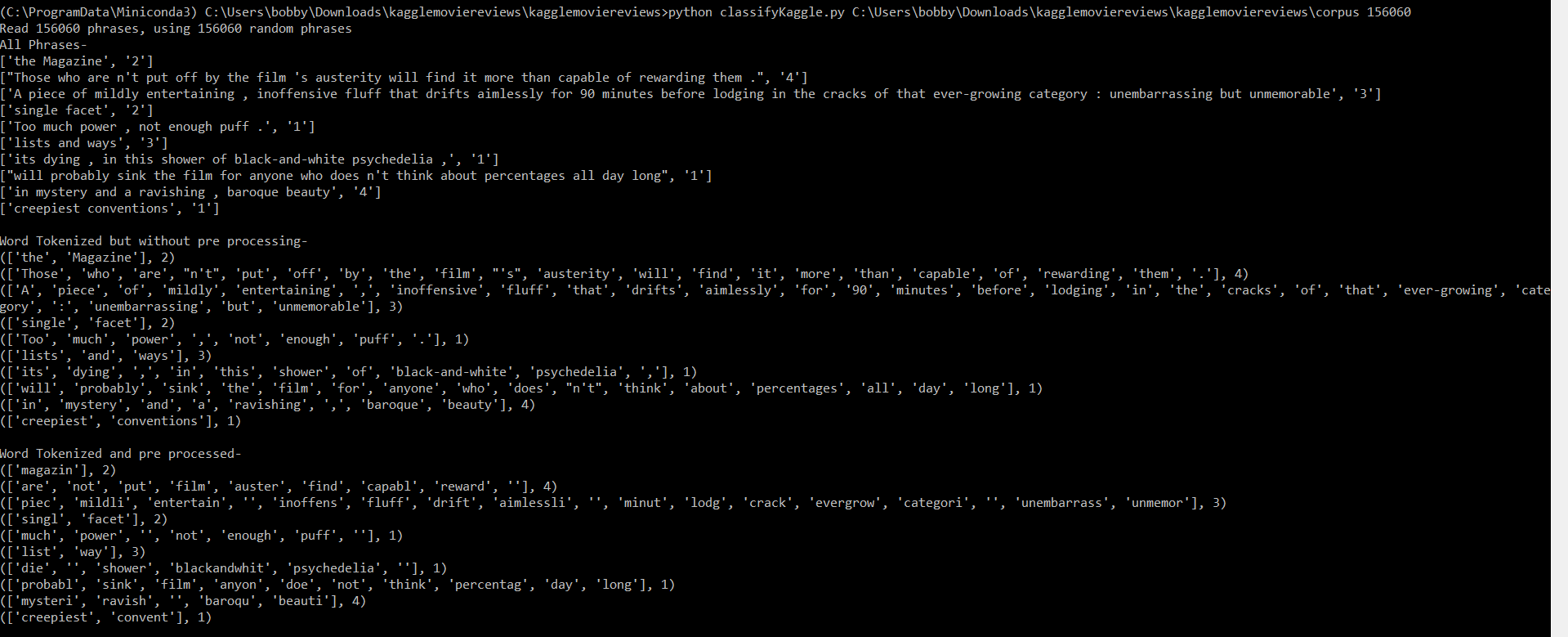
def get\_words(doc):

word\_list = []

for (word, sentiment) in doc:

word\_list.extend(word)

return word\_list



1. **Writing featuresets to a csv file-**

We need to generate csv files of feature set so that they can be later use it with our Weka Classifer. We will use function- writeFeatureSets(featuresets, outpath) defined in save\_features.py file.We will import this file using- import save\_features. I will update writeFeatureSets function in order to convert integer value from 0-4 to corresponding sentiment labels.

def writeFeatureSets(featuresets, outpath):

# open outpath for writing

f = open(outpath, 'w')

# get the feature names from the feature dictionary in the first featureset

featurenames = featuresets[0][0].keys()

# create the first line of the file as comma separated feature names

# with the word class as the last feature name

featurenameline = ''

for featurename in featurenames:

# replace forbidden characters with text abbreviations

featurename = featurename.replace(',','CM')

featurename = featurename.replace("'","DQ")

featurename = featurename.replace('"','QU')

featurenameline += featurename + ','

featurenameline += 'class'

# write this as the first line in the csv file

f.write(featurenameline)

f.write('\n')

for featureset in featuresets:

featureline = ''

for key in featurenames:

try:

featureline += str(featureset[0].get(key,[])) + ','

except KeyError:

continue

if featureset[1] == 0:

featureline += str("strongly negative")

elif featureset[1] == 1:

featureline += str("slightly negative")

elif featureset[1] == 2:

featureline += str("neutral")

elif featureset[1] == 3:

featureline += str("slightly positive")

elif featureset[1] == 4:

featureline += str("strongly positive")

# write each feature set values to the file

f.write(featureline)

f.write('\n')

f.close()

1. **Feature Selection-**

**Bag of words feature:**

**from nltk import FreqDist**

def bag\_of\_words(corpus ,wordcount):

wordlist = nltk.FreqDist(corpus)

word\_features = [w for (w, c) in wordlist.most\_common(wordcount)]

return word\_features

This function collects all the words in the corpus and select some number (depending on wordcount passed as argument) of most frequent words to be the word features. This function will be useful in other features that we are going to define now.

**Bag of words for bigram:**

**from nltk.collocations import \***

def bag\_of\_words\_biagram(wordlist,bigramcount):

bigram\_measures = nltk.collocations.BigramAssocMeasures()

finder = BigramCollocationFinder.from\_words(wordlist,window\_size=3)

finder.apply\_ngram\_filter(lambda w1, w2: len(w1) < 2)

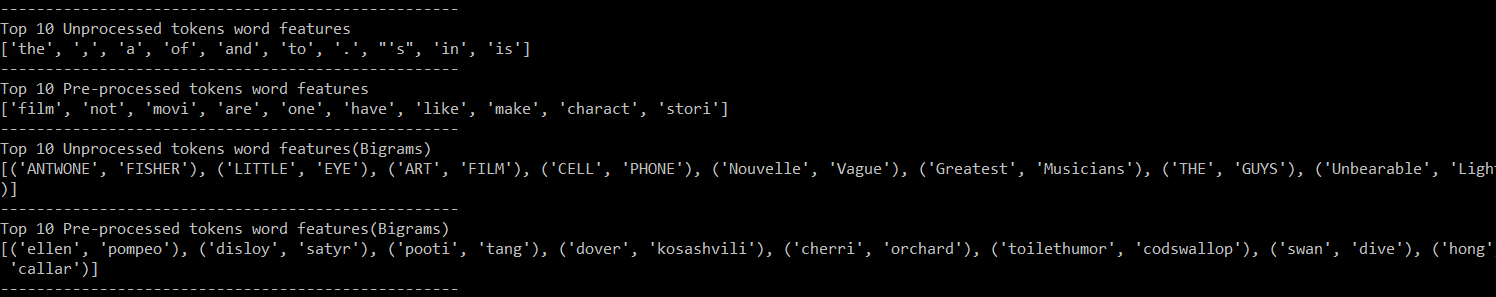
finder.apply\_freq\_filter(3)

bigram\_features = finder.nbest(bigram\_measures.chi\_sq, 3000)

return bigram\_features[:bigramcount]

This function collects all the words in the corpus and select some number (depending on bigramcount passed as argument) of most frequent bigrams. This function will be useful in other features that we are going to define now. We use the chi-squared measure to get bigrams that are informative features. Freq\_filter would remove words that only occurred with a frequency less than 3. Ngram\_filter will filter out bigrams in which the first word’s length is less than 2

Note-It is better to apply this feature to only un-processed tokens as bigram finder must have the words in order. So, this will not produce enough bigrams (with pre-processed tokens) for more accurate results



**Unigram feature (Baseline feature for comparison):**

def unigram\_features(doc, word\_features):

doc\_words = set(doc)

features = {}

for word in word\_features:

featureset['contains(%s)'%word] = (word in doc\_words)

return features

This function returns a dictionary who’s each element is a word (obtained from bag of words function defined earlier) with a Boolean value indicating whether that word occurred in document or not. The feature label will be ‘contains(keyword)’ for each keyword (aka word) in the bag of words set

For example

Unigramsets\_without\_preprocessing -

({'contains(the)': False, 'contains(yet)': False, 'contains(after)': False, 'contains(him)': False, 'contains(take)': False, 'contains(tale)': False, 'contains(years)': False, 'contains(music)': False, 'contains(romantic)': False, 'contains(same)': False, 'contains(documentary)': False, 'contains(subject)': False, 'contains(comes)': False, 'contains(year)': False, 'contains(watching)': False, 'contains(making)': False, 'contains(me)': False, 'contains(worth)': False, 'contains(seem)': False, 'contains(give)': False, 'contains(anything)': False, 'contains(special)': False………

**Bigram feature:**

def bigram\_features(doc,word\_features,bigram\_features):

document\_words = set(doc)

document\_bigrams = nltk.bigrams(doc)

features = {}

for word in word\_features:

features['contains(%s)' % word] = (word in document\_words)

for bigram in bigram\_features:

features['bigram(%s %s)' % bigram] = (bigram in document\_bigrams)

return features

This function takes the list of words in a document as an argument and returns a feature dictionary. It depends on the variables word\_features and bigram\_features

For example

Bigramsets\_without\_preprocessing -

………..'bigram(smallest sensitivities)': False, 'bigram(Bermuda Triangle)': False, 'bigram(Digital stereo)': False, 'bigram(Les Vampires)': False, 'bigram(Movies Ago)': False, 'bigram(Plutonium Circus)': False, 'bigram(craven concealment)': False……….

**Negative features:**

For this feature, I first created my own negative words dictionary and also added processed version negative words (clean text+ stem+lemma) in this dictionary.

Note- I took care of whitespaces in some negative words just like in original corpus so I added ca n’t instead of can’t

negative\_words = ['abysmal','adverse','alarming','angry','annoy','anxious','apathy','appalling','atrocious','awful',

'bad','banal','barbed','belligerent','bemoan','beneath','boring','broken',

'callous','ca n\'t','clumsy','coarse','cold','cold-hearted','collapse','confused','contradictory','contrary','corrosive','corrupt','crazy','creepy','criminal','cruel','cry','cutting','dead','decaying','damage','damaging','dastardly','deplorable','depressed','deprived','deformed''deny','despicable','detrimental','dirty','disease','disgusting','disheveled','dishonest','dishonorable','dismal','distress','do n\'t','dreadful','dreary', 'enraged','eroding','evil','fail','faulty','fear','feeble','fight','filthy','foul','frighten','frightful',

'gawky','ghastly','grave','greed','grim','grimace','gross','grotesque','gruesome','guilty',

'haggard','hard','hard-hearted','harmful','hate','hideous','horrendous','horrible','hostile','hurt','hurtful',

'icky','ignore','ignorant','ill','immature','imperfect','impossible','inane','inelegant','infernal','injure','injurious','insane','insidious','insipid',

'jealous','junky','lose','lousy','lumpy','malicious','mean','menacing','messy','misshapen','missing','misunderstood','moan','moldy','monstrous',

'naive','nasty','naughty','negate','negative','never','no','nobody','nondescript','nonsense','noxious',

'objectionable','odious','offensive','old','oppressive',

'pain','perturb','pessimistic','petty','plain','poisonous','poor','prejudice','questionable','quirky','quit',

'reject','renege','repellant','reptilian','repulsive','repugnant','revenge','revolting','rocky','rotten','rude','ruthless',

'sad','savage','scare','scary','scream','severe','shoddy','shocking','sick',

'sickening','sinister','slimy','smelly','sobbing','sorry','spiteful','sticky','stinky','stormy','stressful','stuck','stupid','substandard','suspect','suspicious',

'tense','terrible','terrifying','threatening',

'ugly','undermine','unfair','unfavorable','unhappy','unhealthy','unjust','unlucky','unpleasant','upset','unsatisfactory',

'unsightly','untoward','unwanted','unwelcome','unwholesome','unwieldy','unwise','upset','vice','vicious','vile','villainous','vindictive',

'wary','weary','wicked','woeful','worthless','wound','yell','yucky',

'are n\'t','cannot','ca n\'t','could n\'t','did n\'t','does n\'t','do n\'t','had n\'t','has n\'t','have n\'t','is n\'t','must n\'t','sha n\'t','should n\'t','was n\'t','were n\'t','would n\'t',

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

This function will pre-process above mentioned negative words dictionary:

def negativewordproc(negativewords):

nwords = []

nwords = clean\_text(negativewords)

nwords = lemmatizer(nwords)

nwords = stemmer(nwords)

return nwords

processnwords = negativewordproc(negative\_words)

negative\_words = negative\_words + processnwords

I look for negation words and negate the word following the negation word. I will go through the document words in order adding the word features, but if the word follows a negation words, change the feature to negated word.

def negative\_features(doc, word\_features, negationwords):

features = {}

for word in word\_features:

features['contains({})'.format(word)] = False

features['contains(NOT{})'.format(word)] = False

# go through document words in order

for i in range(0, len(doc)):

word = doc[i]

if ((i + 1) < len(doc)) and (word in negationwords):

i += 1

features['contains(NOT{})'.format(doc[i])] = (doc[i] in word\_features)

else:

if ((i + 3) < len(doc)) and (word.endswith('n') and doc[i+1] == "'" and doc[i+2] == 't'):

i += 3

features['contains(NOT{})'.format(doc[i])] = (doc[i] in word\_features)

else:

features['contains({})'.format(word)] = (word in word\_features)

return features

For Example

Negativesets\_without\_preprocessing -

({'contains(the)': False, 'contains(NOTthe)': False, 'contains(and)': False, 'contains(NOTand)': False, 'contains(,)': False, 'contains(NOT,)': False, 'contains(of)': False, 'contains(NOTof)': False, 'contains(a)': True, 'contains(NOTa)': False, 'contains(to)': False, 'contains(NOTto)': False………………

**POS feature:**

It runs the default POS tagger (Stanford tagger) on the document and counts 4 types of pos tags to use as features

def POS\_features(doc, word\_features):

document\_words = set(doc)

tagged\_words = nltk.pos\_tag(doc)

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

For example

POSsets\_without\_preprocessing -

…………..'contains(funny)': False, 'contains(comes)': False, 'contains(along)': False, 'contains(occasionally)': False, 'contains(unconventional)': False, 'contains(gutsy)': False, 'contains(perfectly)': False, 'nouns': 1, 'verbs': 0, 'adjectives': 2, 'adverbs': 0}, 1)

Since POS tagger cannot detect normalized form of token we will create a new function for pre-processed form of a sentence that takes un processed form of tokens that will be tagged and then pre-processed.

def POS2\_features(doc,word\_features):

tagged\_words = nltk.pos\_tag(doc)

document\_words = set(doc)

nwords = clean\_text(document\_words)

nwords = rem\_no\_punct(nwords)

nwords = rem\_stopword(nwords)

nwords = lemmatizer(nwords)

nwords = stemmer(nwords)

features = {}

for word in word\_features:

features['contains({})'.format(word)] = (word in nwords)

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

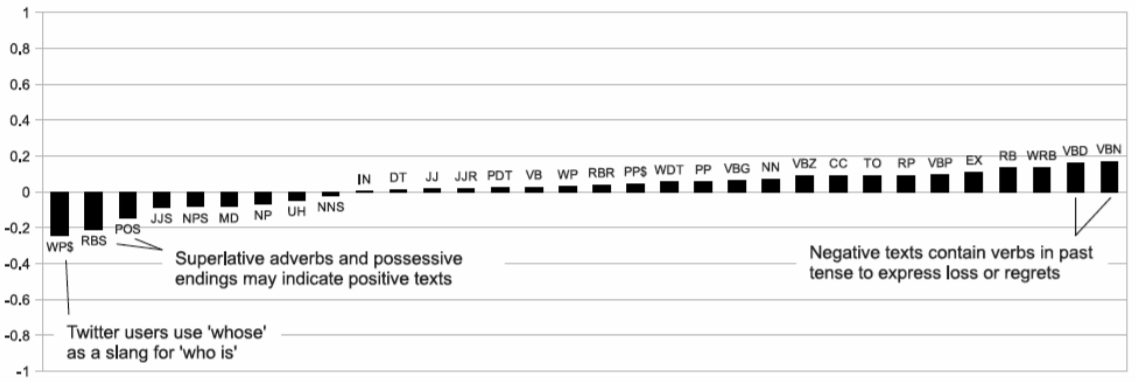
features['nouns'] = numNoun

features['verbs'] = numVerb

features['adjectives'] = numAdj

features['adverbs'] = numAdverb

return features

Note- Based on a study, more past tense verbs mean negative sentiment and more superlative adverb, means positive sentiment so counting POS will also help in sentiment analysis

**Sentiment Lexicon(Subjectivity) feature:**

In order to use this function, we will define one additional function that reads subjectivity words from the subjectivity lexicon file and returns dictionary, where each word is mapped to a list containing the strength and polarity.

def readSubjectivity(path):

flexicon = open(path, 'r')

sldict = { }

for line in flexicon:

fields = line.split() # 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

procword = wordproc(word)

sldict[procword] = [strength, posTag, isStemmed, polarity]

sldict[word] = [strength, posTag, isStemmed, polarity]

return sldict

SL = readSubjectivity(SLpath)

**Note-** I have not imported this function from sentiment\_read\_Subjectivity.py as this function is not same as the one in sentiment\_read\_Subjectivity.py. I have modified it to include pre-processed version of all words in SL for our pre-processed tokens. In order to pre-process individual words in SL dictionary, I have defined another function. This function takes word and returns stemmed and lemmatized version of it.

def wordproc(word):

wnl = nltk.WordNetLemmatizer()

porter = nltk.PorterStemmer()

nwords = wnl.lemmatize(word)

nwords = porter.stem(nwords)

return nwords

This feature function will calculate word counts of subjectivity words. Negative feature will have number of weakly negative words + 2 \* number of strongly negative words. Same way it will count for positive features. It will not count neutral words

def SL\_features(doc, word\_features, SL):

document\_words = set(doc)

features = {}

for word in word\_features:

features['contains({})'.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)

if 'positivecount' not in features:

features['positivecount']=0

if 'negativecount' not in features:

features['negativecount']=0

return features

For example-

Subjectivitysets\_without\_preprocessing -

({'contains(the)': False, 'contains(and)': False, 'contains(,)': False, 'contains(of)': False, 'contains(a)': True, 'contains(to)': False, 'contains(that)': False, "contains('s)": False, 'contains(.)': False, 'contains(in)': False, 'contains(is)': False,……………… 'positivecount': 2, 'negativecount': 0}, 1)

**Sentiment Lexicon(LIWC) feature:**

I have defined another function that will calculate word counts of positive and negative words just like we did subjectivity count earlier.

def liwc\_features(doc, word\_features,poslist,neglist):

doc\_words = set(doc)

features = {}

for word in word\_features:

features['contains({})'.format(word)] = (word in doc\_words)

pos = 0

neg = 0

for word in doc\_words:

if sentiment\_read\_LIWC\_pos\_neg\_words.isPresent(word,poslist):

pos += 1

if sentiment\_read\_LIWC\_pos\_neg\_words.isPresent(word,neglist):

neg += 1

features['positivecount'] = pos

features['negativecount'] = neg

if 'positivecount' not in features:

features['positivecount']=0

if 'negativecount' not in features:

features['negativecount']=0

return features

For example

liwcsets\_without\_preprocessing -

…………'contains(young)': False, 'contains(set)': False, 'contains(conquer)': False, 'contains(online)': False, 'contains(world)': False, 'contains(laptops)': False, 'contains(cell)': False, 'contains(phones)': False, 'contains(sketchy)': False, 'positivecount': 0, 'negativecount': 1}, 2)

I have added pre-processed version of positive words and negative words to their respective dictionary that I got by reading LIWC sentiment lexicon file. For this I have reused function define for pre-processing of negative words dictionary.

import sentiment\_read\_LIWC\_pos\_neg\_words

poslist,neglist = sentiment\_read\_LIWC\_pos\_neg\_words.read\_words()

poslist = poslist+negativewordproc(poslist)

neglist = neglist+negativewordproc(neglist)

**Sentiment lexicon Combination approach:**

If a word is found in positive dictionary of LIWC sentiment lexicon then it will be considered as strongly positive. Similarly, if a word is found in negative dictionary of LIWC sentiment lexicon then it will be considered as strongly negative. Rest of the approach is like subjectivity feature.

def SL\_liwc\_features(doc, word\_features, SL,poslist,neglist):

document\_words = set(doc)

features = {}

for word in word\_features:

features['contains({})'.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 sentiment\_read\_LIWC\_pos\_neg\_words.isPresent(word,poslist):

strongPos += 1

elif sentiment\_read\_LIWC\_pos\_neg\_words.isPresent(word,neglist):

strongNeg += 1

elif 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)

if 'positivecount' not in features:

features['positivecount']=0

if 'negativecount' not in features:

features['negativecount']=0

return features

**Bing Liu’s Opinion Lexicon**

[Bing Liu](http://sentiment.christopherpotts.net/lexicons.html#resources) maintains and freely distributes a sentiment lexicon consisting of lists of strings.

Positive words: 2006

Negative words: 4783

I have defined another function to read this lexicon features and get two dictionaries of positive and negative list so that we can reuse feature function defined for LIWC.

def read\_opinionlexicon():

POLARITY\_DATA\_DIR = os.path.join('polarity-data', 'rt-polaritydata')

POSITIVE\_REVIEWS = os.path.join(POLARITY\_DATA\_DIR, 'rt-polarity-pos.txt')

NEGATIVE\_REVIEWS = os.path.join(POLARITY\_DATA\_DIR, 'rt-polarity-neg.txt')

pos\_features = []

neg\_features = []

for line in open(POSITIVE\_REVIEWS, 'r').readlines()[35:]:

pos\_words = re.findall(r"[\w']+|[.,!?;]", line.rstrip())

pos\_features.append(pos\_words[0])

for line in open(NEGATIVE\_REVIEWS, 'r').readlines()[35:]:

neg\_words = re.findall(r"[\w']+|[.,!?;]", line.rstrip())

neg\_features.append(neg\_words[0])

return pos\_features,neg\_features

Note-Files related to this lexicon can be found in corpus/polarity-data folder

1. **Building Feature set and saving it in csv file-**

**unigramsets\_without\_preprocessing = [(unigram\_features(d, uword\_features), s) for (d, s) in wordtoken]**

**print(" ")**

**print("Unigramsets\_without\_preprocessing -")**

**print(unigramsets\_without\_preprocessing[0])**

**save\_features.writeFeatureSets(unigramsets\_without\_preprocessing,"outputcsv/unigramsets\_without\_preprocessing.csv")**

**print(" ")**

**NLTK Classifiers-**

**Naive Bayes Classifier:**

I am using Naïve Bayes classifier to train and test data with 90 % of data as training set and 10% as test set initially.

def nltk\_naive\_bayes(featuresets,percent):

training\_size = int(percent\*len(featuresets))

train\_set, test\_set = featuresets[training\_size:], featuresets[:training\_size]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

print("Naive Bayes Classifier ")

print("Accuracy : ",nltk.classify.accuracy(classifier, test\_set))

print("Showing most informative features:")

print(classifier.show\_most\_informative\_features(10))

confusionmatrix(classifier,test\_set)

print(" ")

I am also printing confusion matrix to know how many of the actual class labels (the gold standard labels) match with the predicted labels

from nltk.metrics import ConfusionMatrix

def confusionmatrix(classifier\_type, test\_set):

reflist = []

testlist = []

for (features, label) in test\_set:

reflist.append(label)

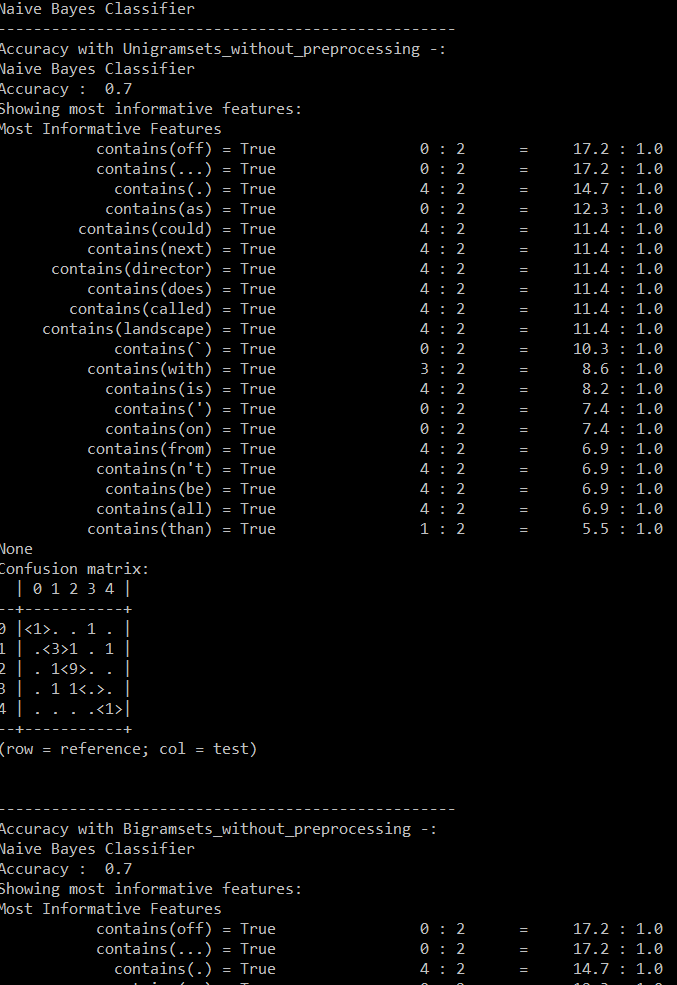
testlist.append(classifier\_type.classify(features))

print("Confusion matrix:")

cm = ConfusionMatrix(reflist, testlist)

print(cm)

Output of Naïve Bayes Classifier will look like this-



**Maximum Entropy Classifier-**

Max Entropy classifier is a probabilistic classifier which belongs to the class of exponential models. Unlike the [Naive Bayes](http://blog.datumbox.com/machine-learning-tutorial-the-naive-bayes-text-classifier/) classifier, the Max Entropy does not assume that the features are conditionally independent of each other.  Max Entropy classifier can be used to solve a large variety of text classification problems such as language detection, topic classification, sentiment analysis and more.

We are going to use three different algorithms of max entropy to train and test our data:

1. Generalized Iterative Scaling (GIS) algorithm
2. Improved Iterative Scaling (IIS)

from nltk.classify import MaxentClassifier

def maximum\_entropy(featuresets,percent):

training\_size = int(percent\*len(featuresets))

train\_set, test\_set = featuresets[training\_size:], featuresets[:training\_size]

classifier1 = MaxentClassifier.train(train\_set, 'GIS', max\_iter = 1)

print("Maximum Entropy Classifier- Generalized Iterative Scaling (GIS) algorithm")

print("Accuracy : ",nltk.classify.accuracy(classifier1, test\_set))

print("Showing most informative features:")

print(classifier1.show\_most\_informative\_features(10))

confusionmatrix(classifier1,test\_set)

print(" ")

classifier2 = MaxentClassifier.train(train\_set, 'IIS', max\_iter = 1)

print("Maximum Entropy Classifier- Iterative Scaling (IIS) algorithm")

print("Accuracy : ",nltk.classify.accuracy(classifier2, test\_set))

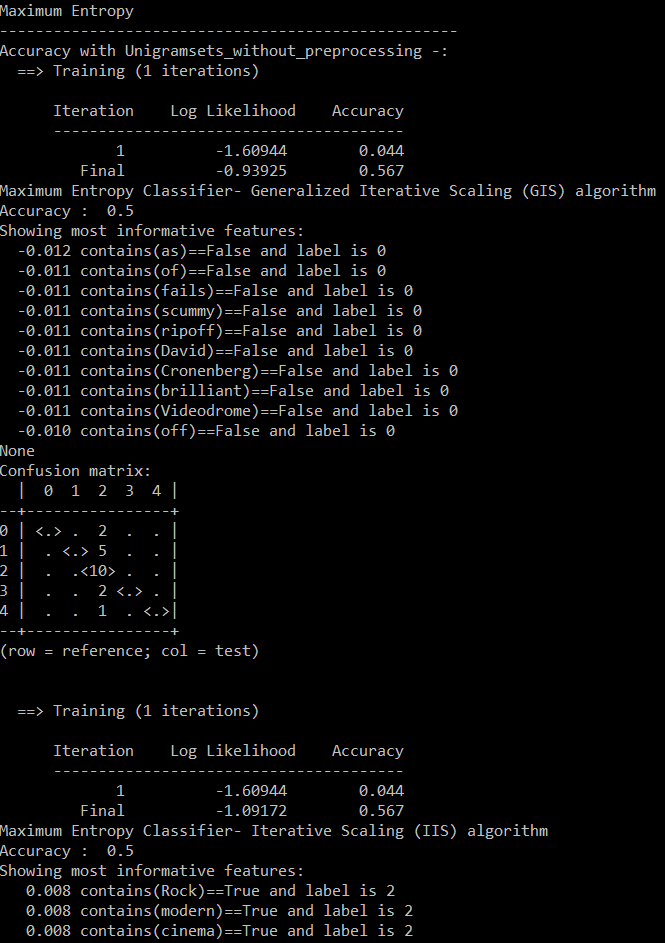
print("Showing most informative features:")

print(classifier2.show\_most\_informative\_features(10))

confusionmatrix(classifier2,test\_set)

print(" ")

Output of Maximum Entropy Classifier will look like this-



**Sci-Kit Learner Classifiers-**

We will also train and test our dataset using 8 algorithms from Sci-kit learner classifiers:

a.) Random Forest

b.) MultinomialNB

c.) BernoulliNB’

d.) Logistic Regressions

e.) SGDClassifer

f.) SVC

g.) Linear SVC

h.) NuSVC

i.) Decision Tree Classifier

from nltk.classify.scikitlearn import SklearnClassifier

from sklearn.naive\_bayes import MultinomialNB, BernoulliNB

from sklearn.linear\_model import LogisticRegression, SGDClassifier

from sklearn.svm import SVC, LinearSVC, NuSVC

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

def sklearn(featuresets,percent):

training\_size = int(percent\*len(featuresets))

train\_set, test\_set = featuresets[training\_size:], featuresets[:training\_size]

classifier1 = SklearnClassifier(MultinomialNB())

classifier1.train(train\_set)

print("ScikitLearn Classifier-MultinomialNB")

print("Accuracy : ",nltk.classify.accuracy(classifier1, test\_set))

print(" ")

classifier2 = SklearnClassifier(BernoulliNB())

classifier2.train(train\_set)

print("ScikitLearn Classifier-BernoulliNB")

print("Accuracy : ",nltk.classify.accuracy(classifier2, test\_set))

print(" ")

classifier3 = SklearnClassifier(DecisionTreeClassifier())

classifier3.train(train\_set)

print("ScikitLearn Classifier-Decision Tree")

print("Accuracy : ",nltk.classify.accuracy(classifier3, test\_set))

print(" ")

classifier4 = SklearnClassifier(LogisticRegression())

classifier4.train(train\_set)

print("ScikitLearn Classifier-LogisticRegression")

print("Accuracy : ",nltk.classify.accuracy(classifier4, test\_set))

print(" ")

classifier5 = SklearnClassifier(SGDClassifier())

classifier5.train(train\_set)

print("ScikitLearn Classifier-SGDCClassifier")

print("Accuracy : ",nltk.classify.accuracy(classifier5, test\_set))

print(" ")

classifier6 = SklearnClassifier(SVC())

classifier6.train(train\_set)

print("ScikitLearn Classifier-SVC")

print("Accuracy : ",nltk.classify.accuracy(classifier6, test\_set))

print(" ")

classifier7 = SklearnClassifier(LinearSVC())

classifier7.train(train\_set)

print("ScikitLearn Classifier-LinearSVC")

print("Accuracy : ",nltk.classify.accuracy(classifier7, test\_set))

print(" ")

classifier8 = SklearnClassifier(NuSVC(nu=0.09))

classifier8.train(train\_set)

print("ScikitLearn Classifier-NuSVC")

print("Accuracy : ",nltk.classify.accuracy(classifier8, test\_set))

print(" ")

classifier9 = SklearnClassifier(RandomForestClassifier())

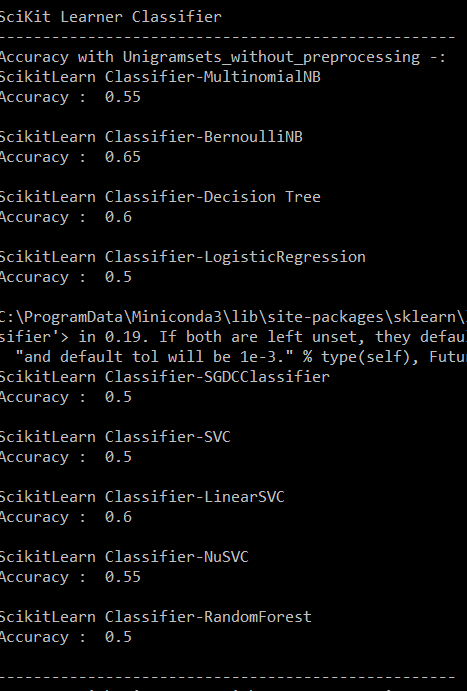
classifier9.train(train\_set)

print("ScikitLearn Classifier-RandomForest")

print("Accuracy : ",nltk.classify.accuracy(classifier9, test\_set))

print(" ")

Output of SciKit Learn Classifier will look like this-



**Single Fold Performances of all classifiers against all feature sets -**

Dataset- limited to 30000 phrases (to avoid memory error)

Bag of words size-500(most frequent words)

90/10 split

1. **Without preprocessing –**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Naïve Bayes | GIS | IIS | Multi Nomial NB | Bernoulli NB | Decision Tree | Logistic Regression | SGDC | SVC | Linear SVC | Nu SVC | Random Forest |
| Unigram | 0.5400 | 0.503 | 0.503 | 0.5516 | 0.5383 | 0.5060 | 0.5563 | 0.5483 | 0.5030 | 0.5536 | 0.1513 | 0.5296 |
| Bigram | 0.5400 | 0.503 | 0.503 | 0.5480 | 0.5430 | 0.5100 | 0.5563 | 0.5493 | 0.5030 | 0.5536 | 0.0663 | 0.5230 |
| POS | 0.5270 | 0.503 | 0.179 | 0.5536 | 0.5283 | 0.4793 | 0.5570 | 0.5190 | 0.5096 | 0.5570 | 0.2206 | 0.5170 |
| Negation | 0.5220 | 0.503 | 0.043 | 0.5303 | 0.5430 | 0.5086 | 0.5630 | 0.5496 | 0.5030 | 0.5606 | 0.4143 | 0.5273 |
| SL | 0.5456 | 0.503 | 0.503 | 0.5663 | 0.5423 | 0.5350 | 0.5680 | 0.5580 | 0.5410 | 0.5670 | 0.1896 | 0.5576 |
| LIWC | 0.5443 | 0.503 | 0.179 | 0.5526 | 0.5450 | 0.5260 | 0.5663 | 0.5420 | 0.5400 | 0.5606 | 0.1323 | 0.5386 |
| SL+ LIWC | 0.5500 | 0.503 | 0.179 | 0.5703 | 0.5443 | 0.5416 | 0.5690 | 0.5310 | 0.5573 | 0.5663 | 0.1773 | 0.5510 |
| Opinion | 0.5443 | 0.503 | 0.179 | 0.5526 | 0.545 | 0.5250 | 0.5663 | 0.5463 | 0.5400 | 0.5603 | 0.1323 | 0.5530 |

1. **With preprocessing –**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Naïve Bayes | GIS | IIS | Multi Nomial NB | Bernoulli NB | Decision Tree | Logistic Regression | SGDC | SVC | Linear SVC | Nu SVC | Random Forest |
| Unigram | 0.5610 | 0.503 | 0.503 | 0.5546 | 0.5603 | 0.5570 | 0.5600 | 0.5556 | 0.5030 | 0.5563 | 0.1696 | 0.5596 |
| Bigram | 0.5606 | 0.503 | 0.503 | 0.5533 | 0.5580 | 0.5600 | 0.5600 | 0.5520 | 0.5030 | 0.5556 | 0.1220 | 0.5633 |
| POS | 0.551 | 0.503 | 0.503 | 0.5526 | 0.5486 | 0.5170 | 0.5633 | 0.5516 | 0.5050 | 0.5626 | 0.1166 | 0.5320 |
| Negation | 0.5306 | 0.503 | 0.043 | 0.5326 | 0.5333 | 0.5556 | 0.5530 | 0.5520 | 0.5030 | 0.5546 | 0.2516 | 0.5590 |
| SL | 0.5716 | 0.503 | 0.503 | 0.5613 | 0.5613 | 0.5556 | 0.5773 | 0.5626 | 0.5350 | 0.5650 | 0.129 | 0.5746 |
| LIWC | 0.5576 | 0.503 | 0.503 | 0.5520 | 0.5546 | 0.5590 | 0.5670 | 0.5543 | 0.5396 | 0.5633 | 0.132 | 0.5663 |
| SL+ LIWC | 0.5706 | 0.503 | 0.503 | 0.5593 | 0.5616 | 0.5560 | 0.5746 | 0.5320 | 0.5333 | 0.5650 | 0.216 | 0.5676 |
| Opinion | 0.5443 | 0.503 | 0.503 | 0.5520 | 0.5546 | 0.5580 | 0.5670 | 0.5626 | 0.5396 | 0.5633 | 0.1320 | 0.5700 |

**Observations-**

**I will consider unigram feature (naïve bayes) without preprocessing accuracy i.e. 0.54 as baseline for comparison**

We will use single fold results to drop some classifiers from cross validation testing especially ones whose accuracy remained low and did not show any improvement for any feature set.

1. Our maximum entropy classifiers (GIS and IIS) were just able to achieve 50% accuracy. Moreover, they did not show any improvement with any additional feature set. In fact, accuracy remained same for both processed and pre-processed version of document. So, we will drop this classifier for cross validation test.
2. SVC classifier also showed similar results, so we will drop that also.
3. NuSVC classidfier never achieved more than 25% accuracy for any featureset so we will drop this also.
4. Bigram featureset achieved similar accuracy as unigram featureset. Whereas POS and Negation featureset frequency was below unigramset
5. Sentiment lexicons showed slight improvement in performance compared to unigrams
6. Overall, Preprocessed version achieved higher accuracy compared to un processed version.

**Note- Green color depicts better accuracy compared to baseline**

**Yellow color depicts similar level of accuracy**

**Red color depicts accuracy below standards**

**I have included screenshot of every result mentioned above as evidence in corpus/Single fold output folder**

**Cross Validation-**

I defined cross-validation functions for every classifier so that they can be trained in multifold and also calculated accuracy, fscore, recall and precision

def naive\_bayes(num\_folds, featuresets, label\_list):

subset\_size = int(len(featuresets)/num\_folds)

# overall gold labels for each instance (reference) and predicted labels (test)

reflist = []

testlist = []

accuracy\_list = []

print("Naive Bayes Classifier")

# iterate over the folds

for i in range(num\_folds):

print('Start Fold', i)

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)

# add the gold labels and predicted labels for this round to the overall lists

for (features, label) in test\_this\_round:

reflist.append(label)

testlist.append(classifier.classify(features))

print('Done with cross-validation')

# call the evaluation measures function

print('mean accuracy-', sum(accuracy\_list) / num\_folds)

(precision\_list, recall\_list) = eval\_measures(reflist, testlist, label\_list)

print\_evaluation (precision\_list, recall\_list, label\_list)

print(" ")

Similarly, I have defined functions for every classifier.I have also modified original f-score and print\_evaluation functions to take care of DivisionByZero error and when precision type is None.

def eval\_measures(reflist, testlist, label\_list):

#initialize sets

# for each label in the label list, make a set of the indexes of the ref and test items

# store them in sets for each label, stored in dictionaries

# first create dictionaries

ref\_sets = {}

test\_sets = {}

# create empty sets for each label

for lab in label\_list:

ref\_sets[lab] = set()

test\_sets[lab] = set()

# get gold labels

for j, label in enumerate(reflist):

ref\_sets[label].add(j)

# get predicted labels

for k, label in enumerate(testlist):

test\_sets[label].add(k)

# lists to return precision and recall for all labels

precision\_list = []

recall\_list = []

#compute precision and recall for all labels using the NLTK functions

for lab in label\_list:

precision\_list.append ( precision(ref\_sets[lab], test\_sets[lab]))

recall\_list.append ( recall(ref\_sets[lab], test\_sets[lab]))

return (precision\_list, recall\_list)

# This function computes F-measure (beta = 1) from precision and recall

def Fscore (precision, recall):

print(precision)

print(recall)

if (precision == 0.0) and (recall == 0.0 ):

return 0.0

else:

return (2.0 \* precision \* recall) / (precision + recall)

# this function prints precision, recall and F-measure for each label

def print\_evaluation(precision\_list, recall\_list, label\_list):

fscore=[]

num\_folds=0

num=0

for index, lab in enumerate(label\_list):

num +=1

if precision\_list[index] is None:

precision\_list[index]=0.0

if recall\_list[index] is None:

recall\_list[index]=0.0

fscore.append(Fscore(precision\_list[index],recall\_list[index]))

if fscore[num\_folds]==0:

num-=1

num\_folds += 1

print('average precision', sum(precision\_list)/num\_folds)

print('average recall ', sum(recall\_list)/num\_folds)

print('F-score ',sum(fscore)/num)

**Cross Validation results-**

Dataset- limited to 50000 phrases (to avoid memory error)

Bag of words size-500(most frequent words)

90/10 split

No. of folds- 5

Note- We will use complete dataset in Weka classifier and some of the current classifiers as it takes long time to train and test entire dataset. We will shortlist classifiers here based on certain experiment like we did in single fold run so that we are left with few classifiers for training on Weka classifier and some of current classifiers

**Unigram Feature sets:**

|  |  |
| --- | --- |
| Without Pre-processing | With Pre-processing |
|  |  |

**Bigram Feature sets:**

|  |  |
| --- | --- |
| Without Pre-processing | With Pre-processing |
|  |  |

**POS Feature sets:**

|  |  |
| --- | --- |
| Without Pre-processing | With Pre-processing |
|  |  |

**Negation Feature sets:**

|  |  |
| --- | --- |
| Without Pre-processing | With Pre-processing |
|  |  |

**Subjectivity Feature sets:**

|  |  |
| --- | --- |
| Without Pre-processing | With Pre-processing |
|  |  |

**LIWC Feature sets:**

|  |  |
| --- | --- |
| Without Pre-processing | With Pre-processing |
|  |  |

**SL + LIWC Feature sets:**

|  |  |
| --- | --- |
| Without Pre-processing | With Pre-processing |
|  |  |

**Opinion Lexicon Feature sets:**

|  |  |
| --- | --- |
| Without Pre-processing | With Pre-processing |
|  |  |

**Summary(Accuracy)-**

Dataset- limited to 50000 phrases (to avoid memory error)

Bag of words size-500(most frequent words)

90/10 split

No. of folds- 5

1. Without pre-processing

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Naïve Bayes | Multi Nomial NB | Bernoulli NB | Decision Tree | Logistic Regression | SGDC | Linear SVC | Random Forest | **Average**  **(Feature-wise)** |
| Unigram | 0.5312 | 0.5529 | 0.5305 | 0.5145 | 0.5600 | 0.5494 | 0.5582 | 0.5422 | **0.5423** |
| Bigram | 0.5312 | 0.5521 | 0.5332 | 0.5146 | 0.5600 | 0.5446 | 0.5581 | 0.5446 | **0.5423** |
| POS | 0.5200 | 0.5499 | 0.5238 | 0.5015 | 0.5609 | 0.5395 | 0.5588 | 0.5357 | **0.5362** |
| Negation | 0.5311 | 0.5365 | 0.5458 | 0.5131 | 0.5615 | 0.5492 | 0.5598 | 0.5478 | **0.5431** |
| SL | 0.5425 | 0.5688 | 0.5395 | 0.5414 | 0.5706 | 0.5514 | 0.5677 | 0.5666 | **0.5560** |
| LIWC | 0.5417 | 0.5594 | 0.5407 | 0.5313 | 0.5677 | 0.5532 | 0.5634 | 0.5586 | **0.5520** |
| SL+ LIWC | 0.5441 | 0.5700 | 0.5405 | 0.5416 | 0.5707 | 0.5498 | 0.5680 | 0.5687 | **0.5566** |
| Opinion | 0.5417 | 0.5594 | 0.5407 | 0.5330 | 0.5677 | 0.5528 | 0.5634 | 0.5592 | **0.5522** |
| **Average**  **(Classifier-wise)** | **0.5354** | **0.5561** | **0.5368** | **0.5238** | **0.5648** | **0.5487** | **0.5621** | **0.5529** |  |

1. With pre-processing

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Naïve Bayes | Multi Nomial NB | Bernoulli NB | Decision Tree | Logistic Regression | SGDC | Linear SVC | Random Forest | **Average**  **(Feature-wise)** |
| Unigram | 0.5545 | 0.5527 | 0.5540 | 0.5679 | 0.5609 | 0.5516 | 0.5595 | 0.5707 | **0.5589** |
| Bigram | 0.5545 | 0.5512 | 0.5547 | 0.5679 | 0.5610 | 0.5513 | 0.5596 | 0.5700 | **0.5587** |
| POS | 0.5435 | 0.5549 | 0.5485 | 0.5293 | 0.5625 | 0.5388 | 0.5609 | 0.5495 | **0.5484** |
| Negation | 0.5309 | 0.5407 | 0.5457 | 0.5675 | 0.5619 | 0.5529 | 0.5617 | 0.5699 | **0.5539** |
| SL | 0.5604 | 0.5645 | 0.5580 | 0.5660 | 0.5685 | 0.5480 | 0.5646 | **0.5781** | **0.5635** |
| LIWC | 0.5561 | 0.5590 | 0.5555 | 0.5673 | 0.5647 | 0.5601 | 0.5617 | 0.5760 | **0.5625** |
| SL+ LIWC | 0.5612 | 0.5644 | 0.5571 | 0.5650 | 0.5689 | 0.5438 | 0.5643 | 0.5768 | **0.5626** |
| Opinion | 0.5561 | 0.5590 | 0.5555 | 0.5678 | 0.5647 | 0.5503 | 0.5617 | 0.5732 | **0.5610** |
| **Average**  **(Classifier-wise)** | **0.5521** | **0.5558** | **0.5536** | **0.5623** | **0.5641** | **0.5496** | **0.5617** | **0.5705** |  |

**Feature set comparison-**

**Unigram Vs Bigram Feature set: No improvement in accuracy**

Bigram accuracy was almost equal to its corresponding unigram feature set for both pre-processed as well un processed tokens and for all classifiers.

**Unigram Vs POS Feature set: Accuracy declined**

Average POS accuracy was 1% lower than its corresponding unigram feature set for both pre-processed as well un processed tokens. We need to classify parts of speech in more specific categories rather than general noun, verb, adverb and adjectives especially for sentiment analysis in order to obtain higher accuracies.

**Unigram Vs Negation Feature set: No improvement in accuracy**

Negation accuracy was almost equal to its corresponding unigram feature set for both pre-processed as well un processed tokens and for all classifiers.

**Unigram Vs Lexicon Feature sets: Improvement in accuracy**

* Average subjectivity accuracy was 1.35% higher than its corresponding unigram feature set for un-processed tokens. Average subjectivity accuracy was 0.45% higher than its corresponding unigram feature set for pre-processed tokens. In fact, Random Forest classifier produced best accuracy (0.5781) using subjectivity feature set on pre-processed version.
* Average LIWC accuracy was 1.00% higher than its corresponding unigram feature set for un-processed tokens. Average LIWC accuracy was 0.35% higher than its corresponding unigram feature set for pre-processed tokens.
* Average SL + LIWC accuracy was 1.40% higher than its corresponding unigram feature set for un-processed tokens. Average SL + LIWC accuracy was 0.36% higher than its corresponding unigram feature set for pre-processed tokens.
* Average opinion lexicon accuracy was 1.01% higher than its corresponding unigram feature set for un-processed tokens. Average opinion lexicon accuracy was 0.25% higher than its corresponding unigram feature set for pre-processed tokens.

**Classifier comparison-**

* BernoulliNB classifier average accuracy was almost equal to its corresponding average accuracy of Naïve Bayes classifier for both pre-processed as well unprocessed tokens.
* Logistic Regression, LinearSVC, Multi nomialNB and Random Forest classifiers produced better accuracy than its corresponding Naïve Bayes classifier for both pre-processed as well as un-processed tokens.
* DecisionTree and SGDC classifier performance varied compared to its Naïve Bayes counterpart.

DecisionTree classifier produced better performance than Naive Bayes for pre-processed version whereas its performance was lower than Naive Bayes for unprocessed version. SGDC classifier produced better performance than Naive Bayes for un-processed version whereas its performance was lower than Naive Bayes for pre-processed version.

**Pre-processed Vs Un-processed-**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Accuracy** | **F-measure** | **Precision** | **Recall** |
| **Un-processed** | 0.5475 | 0.3467 | 0.4281 | 0.3365 |
| **Pre-processed** | **0.5587** | 0.3455 | **0.4497** | 0.3362 |

**Accuracy** - It is ratio of correctly predicted observation to the total observations. If we have high accuracy, then our model is best. Yes, accuracy is a great measure but only when you have symmetric datasets where values of false positive and false negatives are almost same. Therefore, we have to look at other parameters to evaluate the performance of your model. Our pre-processed version achieved higher accuracy compared to un processed version.

**Precision** - Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. High precision relates to the low false positive rate. We achieved higher precision with pre-processed version

Precision = TP/TP+FP

**Recall**- Recall is the ratio of correctly predicted positive observations to the all observations in actual class. We achieved similar levels of recall (0.336)

Recall = TP/TP+FN

**F1 score** - F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. F1 is usually more useful than accuracy, especially if you have an uneven class distribution. Accuracy works best if false positives and false negatives have similar cost. If the cost of false positives and false negatives are very different, it’s better to look at both Precision and Recall. We achieved similar levels of F1-measure (0.34)

F1 Score = 2\*(Recall \* Precision) / (Recall + Precision)

**Observation-** With higher accuracy and precision, pre-processed version of dataset is more suitable for our sentiment analysis and classification. This is because pre-processed version removed unnecessary words**.**

**Different size of vocabulary-**

We will use pre-processed version of dataset and following classifier for this study:

1. Naïve Bayes (Baseline)
2. Random Forest (Best accuracy)
3. SGDC (Worst accuracy)

Also, we will limit our study on following feature sets:

1. Unigram(baseline)
2. Subjectivity + LIWC (best accuracy)
3. Opinion (accuracy between Unigram and Subjectivity)

Other parameters will remain same i.e. 90/10 split, 5 folds

**For Vocabulary size-300**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Naïve Bayes | SGDC | Random Forest |
| Unigram | 0.5473 | 0.5387 | 0.5638 |
| SL+ LIWC | 0.5545 | 0.5362 | 0.5739 |
| Opinion | 0.5486 | 0.5442 | 0.5680 |
| **Average** | **0.5528** | | |

**For Vocabulary size-500**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Naïve Bayes | SGDC | Random Forest |
| Unigram | 0.5545 | 0.5516 | 0.5707 |
| SL+ LIWC | 0.5612 | 0.5438 | 0.5768 |
| Opinion | 0.5561 | 0.5503 | 0.5732 |
| **Average** | **0.5598** | | |

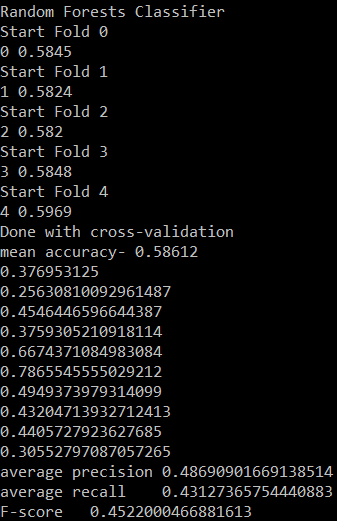
**For Vocabulary size -1000**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Naïve Bayes | SGDC | Random Forest |
| Unigram | 0.5664 | 0.5662 | 0.5799 |
| SL+ LIWC | 0.5779 | 0.5665 | 0.5854 |
| Opinion | 0.5699 | 0.5718 | 0.5831 |
| **Average** | **0.5741** | | |

**For Vocabulary size -2000**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Naïve Bayes | SGDC | Random Forest |
| Unigram | 0.5747 | 0.5780 | 0.5845 |
| SL+ LIWC | 0.5803 | 0.5713 | **0.5861** |
| Opinion | 0.5790 | 0.5784 | 0.5856 |
| **Average** | **0.5797** | | |

**Observation**-Increasing vocabulary size also increases accuracy. We also got our new best accuracy of 0.5861 with SL\_LIWC feature set (best feature set as we predicted earlier) and random forest classifier (best classifier as we predicted earlier)



**Different size of dataset-**

Till now we were limiting our dataset size to 30000 phrases for single fold, 50000 phrases for cross validation (5 fold) now we will train and test on entire dataset (1,56,060 phrases) with other parameters (best we found so far) as follows:

Vocabulary size: 2000

Feature sets: SL+LIWC (Best till now), Opinion (in between other two), Unigram(Baseline)

Classifiers: Naïve Bayes(Baseline), SGDC (worst accuracy), Random Forest(Best)

Version: Pre-processed

No. of folds- 5

Split ratio :90/10

**For Dataset size- 30,000 phrases**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Naïve Bayes | SGDC | Random Forest |
| Unigram | 0.5413 | 0.5297 | 0.5549 |
| SL+ LIWC | 0.5502 | 0.5335 | 0.5620 |
| Opinion | 0.5446 | 0.5302 | 0.5595 |
| **Average** | **0.5451** | | |

**For Dataset size- 50,000 phrases**

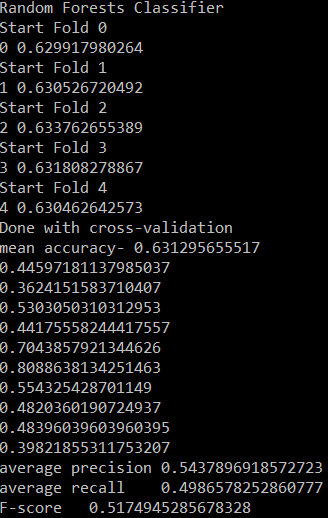
|  |  |  |  |
| --- | --- | --- | --- |
|  | Naïve Bayes | SGDC | Random Forest |
| Unigram | 0.5473 | 0.5387 | 0.5638 |
| SL+ LIWC | 0.5545 | 0.5362 | 0.5739 |
| Opinion | 0.5486 | 0.5442 | 0.5680 |
| **Average** | **0.5528** | | |

**For Dataset size-1,56,060 phrases**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Naïve Bayes | SGDC | Random Forest |
| Unigram | 0.5886 | 0.5793 | 0.6282 |
| SL+ LIWC | 0.5944 | 0.5847 | **0.6312** |
| Opinion | 0.5912 | 0.5837 | 0.6294 |
| **Average** | **0.6011** | | |

Note- I have added screenshot for above experiment (entire dataset part) in corpus/Multifold output folder. Also in order to train entire dataset, I had to run code feature wise by selecting one feature at a time and commenting other features. This avoided memory errors.

**Observation-** Increasing dataset size also increases accuracy. We also got our new best accuracy of 0.6312 with SL\_LIWC feature set (best feature set as we predicted earlier) and random forest classifier (best classifier as we predicted earlier).

****

**WEKA Classifier-**

We have already covered Sci kit classifier and one extra sentiment lexicon (opinion lexicon) for our advanced level tasks but it would be interesting to try GUI based classifier like WEKA especially on our optimized parameters that we found till now as mentioned below-

Vocabulary size: 2000

Dataset: all phrases (156060 phrases)

Feature sets: SL+LIWC, Opinion, Unigram

Version: Pre-processed

Note-Only SL\_LIWCfeaturesets\_with\_preprocessing, opinionsets\_with\_preprocessing, and unigramfeaturesets\_with\_preprocessing has all phrases in csv format and rest all csv files are built for 50,000 phrases.

**Experiment 1 - Weka Classifier Vs Others (Cross validation 5 folds)**

1. **Weka Classifier Vs NLTK**

|  |  |  |
| --- | --- | --- |
|  | NLTK  Naïve Bayes | Weka  Naïve Bayes |
| Unigram | 0.5886 | 0.5870 |
| SL+ LIWC | 0.5944 | 0.5792 |
| Opinion | 0.5912 | 0.5821 |
| **Average** | **0.5914** | **0.5827** |

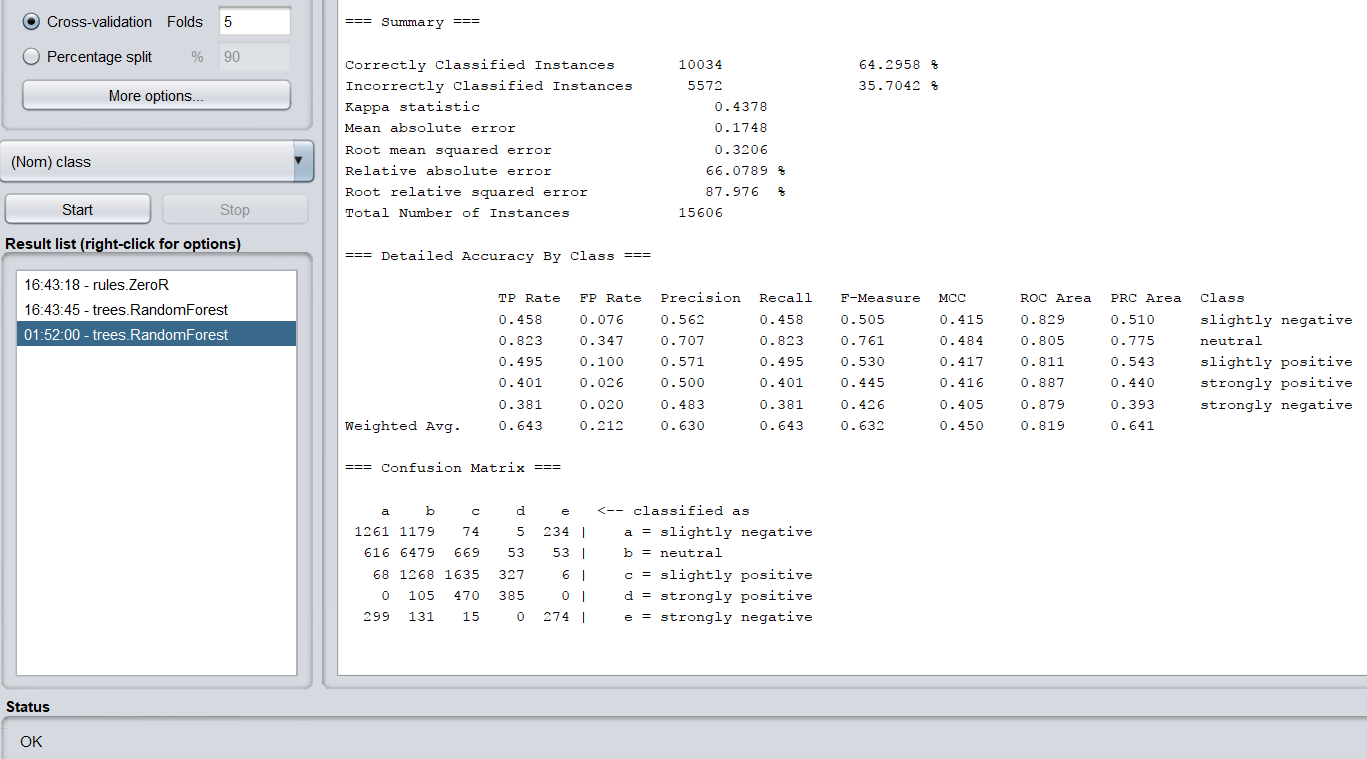
**Observation-**NLTK Naive Bayes outperformed Weka naive Bayes classifier with higher accuracy

1. **Weka Classifier Vs Sci Kit learn**

|  |  |  |
| --- | --- | --- |
|  | Weka  Random Forest | Sci Kit  Random Forest |
| Unigram | 0.6325 | 0.6282 |
| SL+ LIWC | **0.6429** | 0.6312 |
| Opinion | 0.6325 | 0.6294 |
| **Average** | **0.6359** | **0.6296** |

**Observation-**Weka’s Random Forest version outperformed its counterpart-NLTK Random Forest classifier with higher accuracy. **We also obtained our new best accuracy score of 0.6429 again with or combined SL+LIWC feature set and random forest classifier with only difference is being that this time Random Forest version is from Weka**

Note- You can find screenshot of Weka classifiers in corpus/weka classifer folder

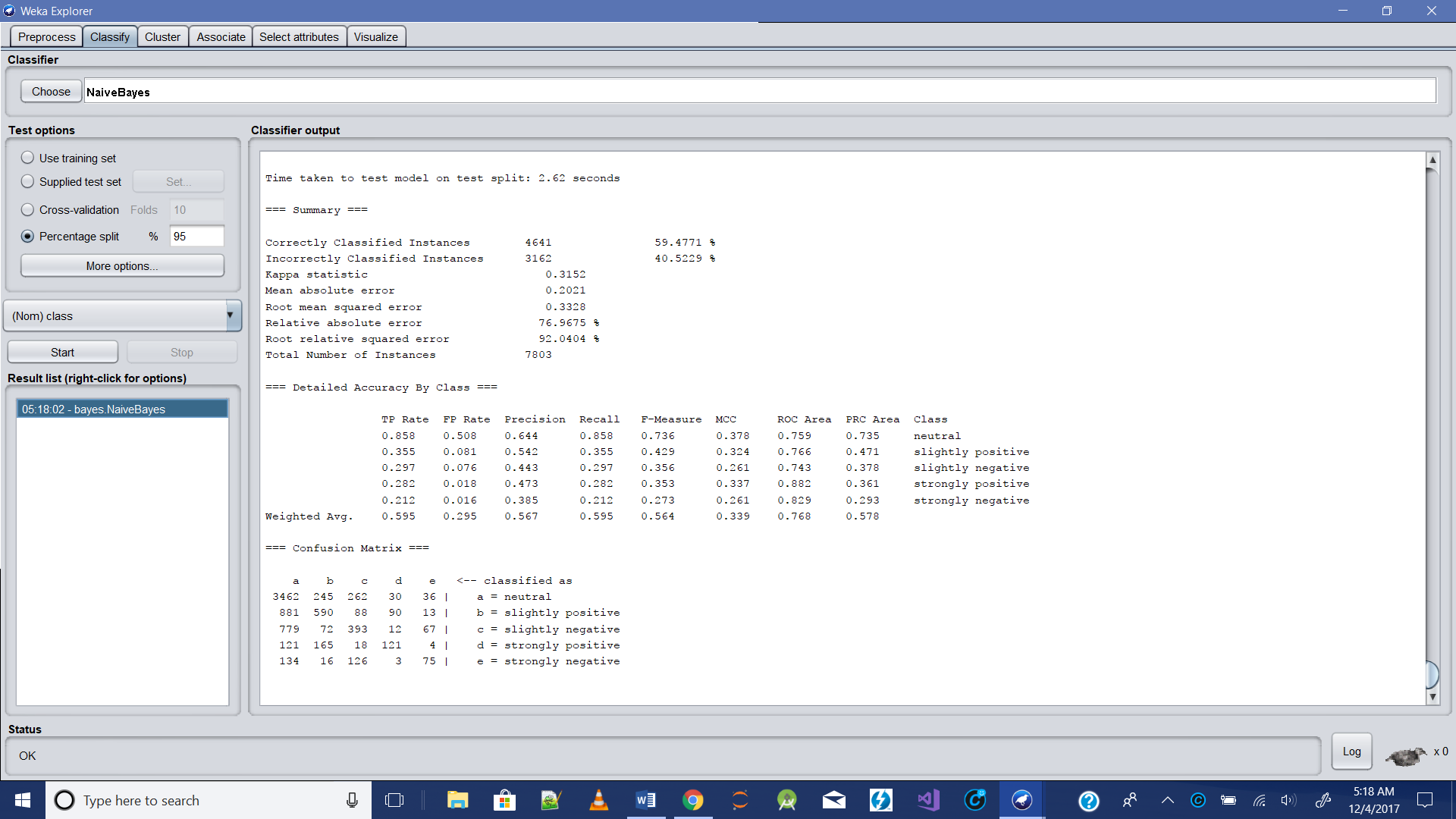


**Experiment 2- Varying split ratio**

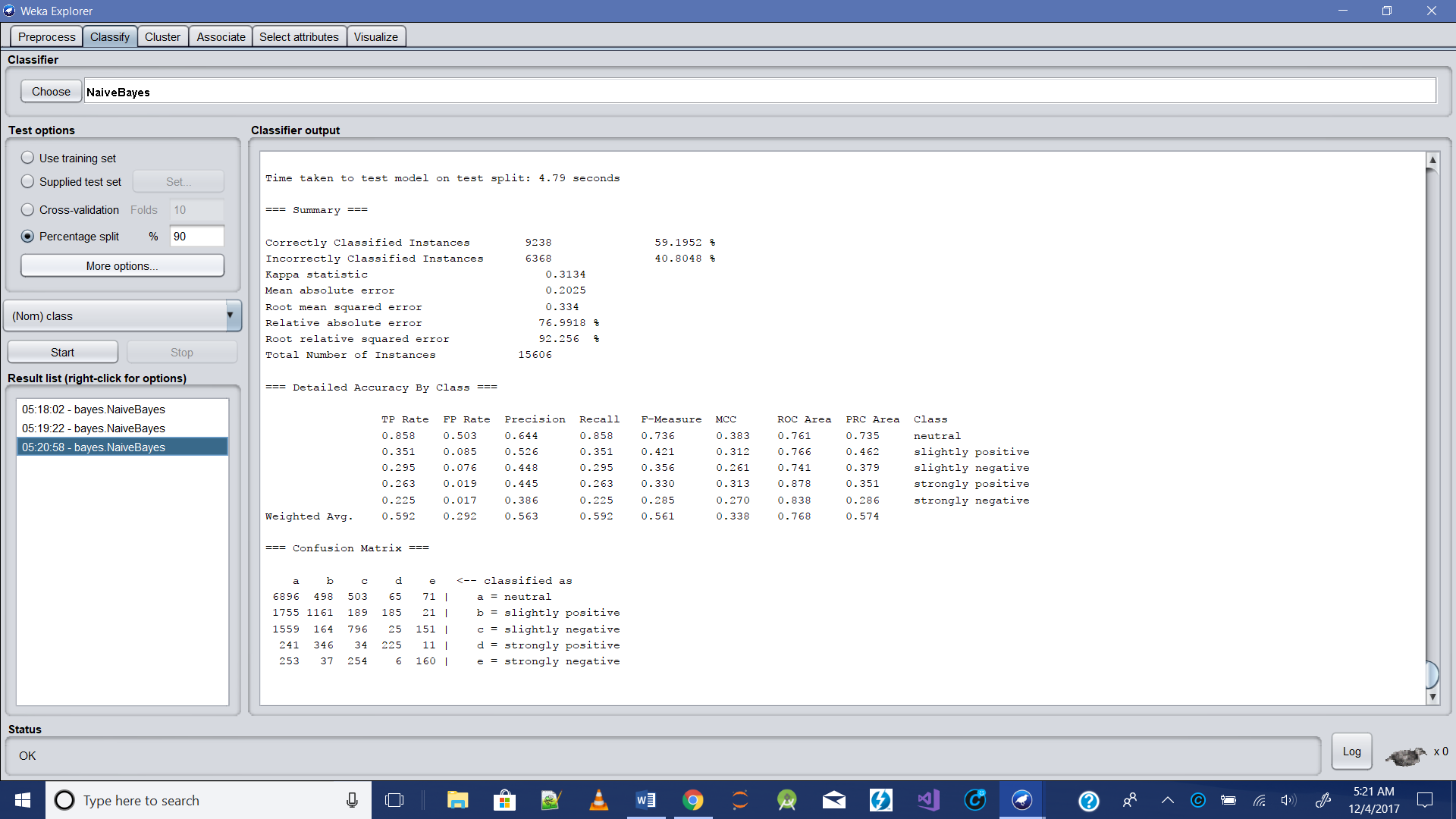
Classifier-Naive Bayes

Feature set-Unigram

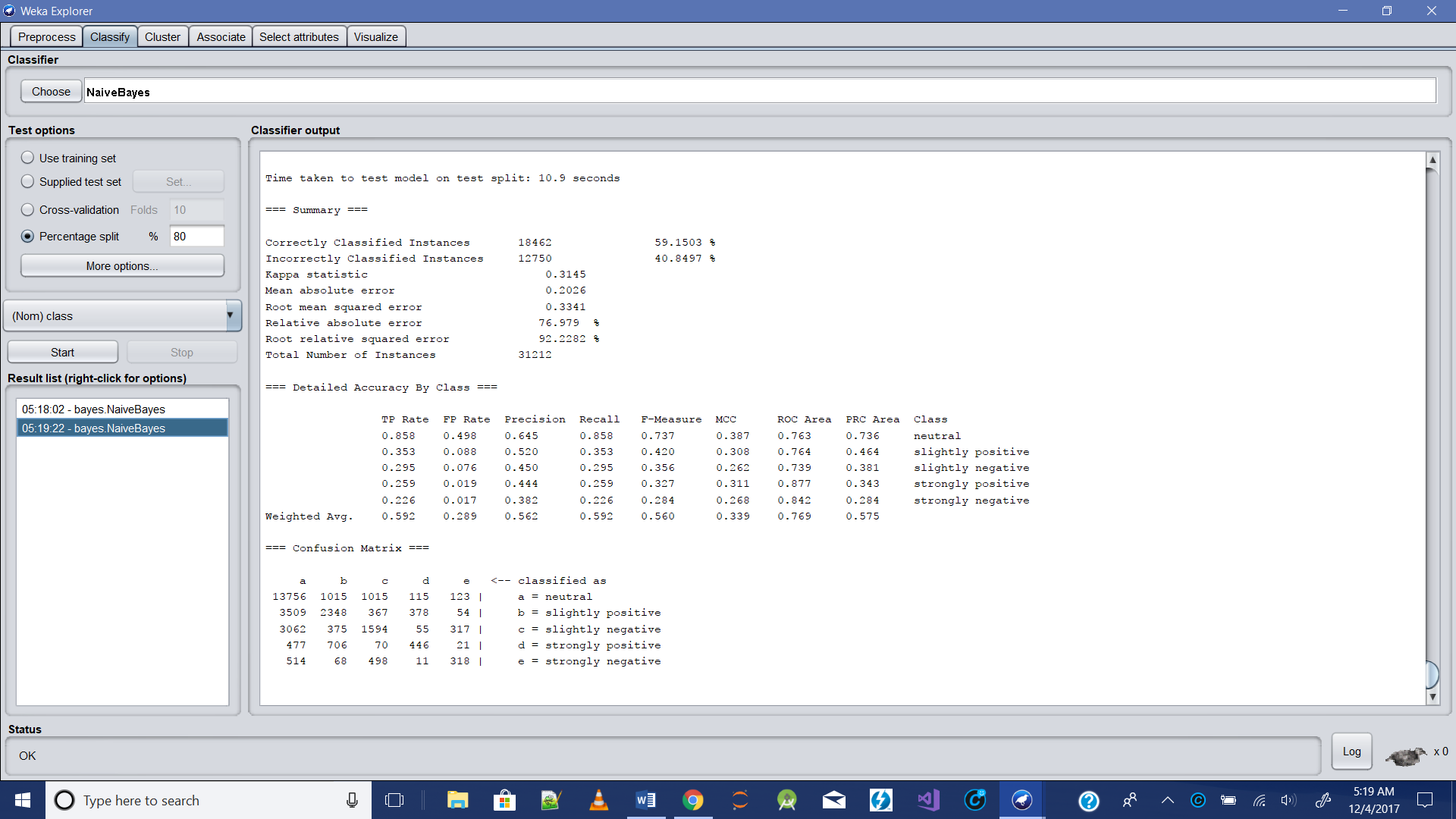
a.)95/5 ratio



b.)90/10



b.)80/20



**Observation-**Reducing training size of data, reduces accuracy(slightly).

**Summary**

I obtained **best accuracy of 0.6429 on a scale of 1** with Weka’s random forest and SL+LIWC feature set. Although I tried to cover as many experiments as possible but still I had to limit my study to certain parameters because some experiments took 9+ hours to complete as the size of dataset was very big.