Sentimental Analysis and Classification of Amazon Reviews

- **‘pos’ and ‘neg’ tags:-**

First of all I tried 3 methods to tag for positive and negative sentences. First method was through POS taggers. Second is from the featuresets which was in the lab. And the last one was included featureset and the some modification as more words to compare and get better performance for tagging the sentences.

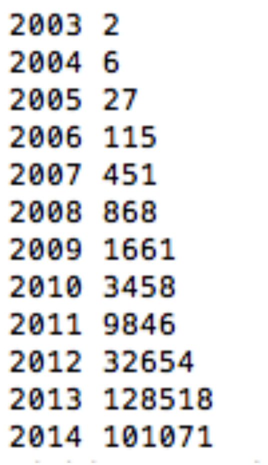
One the first method the sentences are tagged with the sentences are split and tagged with POS Stanford tagger. After that in compared with the dictionary of words with has a value as ‘positive’ and ‘negative’. I used a file which has 8000 words which are tagged as ‘pos’ and ‘neg’.

After the whole process is done I got much less negative sentences than the positive ones. And overall because we don’t use POS tagger for Sentimental Analysis. So I just did it to see the output.

I have attached the output files of this in the Output folder.

For the second approach I did the same as we learned in lab. But in that the words as most_common as 2000 words from the movie review corpus. I modified some words in the corpus as I talked with the professor and TA(Jenna) and compared 3000 words as most_common in the corpus. It gave fairly better performance than the previous one.

For tagging the words first I fetched the reviewtext and reviewtime from the Clothing_shoes_jewelry.txt. As the professor told I fetched the review texts as the years in the date.



```
2003 2
2004 6
2005 27
2006 115
2007 451
2008 868
2009 1661
2010 3458
2011 9846
2012 32654
2013 128518
2014 101071
```

As you can see it, the date range is 2003 to 2014.

I selected 2009 and 2010 both for reviewtexts.

I executed both the set of review sets separately and one as combined.
For the sentences to tag:-

- lower case
- Tokenize (NLTK Word tokenizer)
- Add featuresets
- Used Negation words

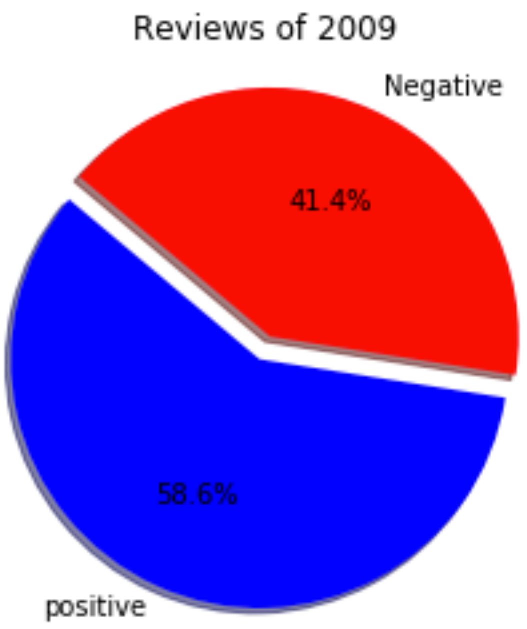
For 2009 first I got the length of Positive sentences and negative sentences as 974 and 687 respectively.

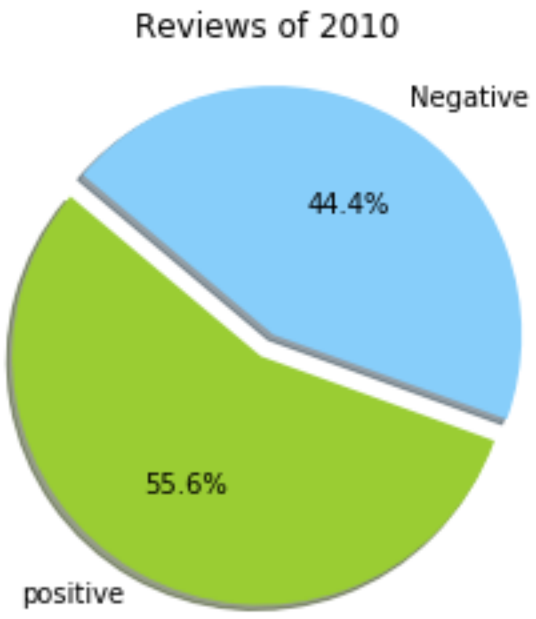
For 2010 first I got the length of Positive sentences and negative sentences as 1925 and 1538 respectively.

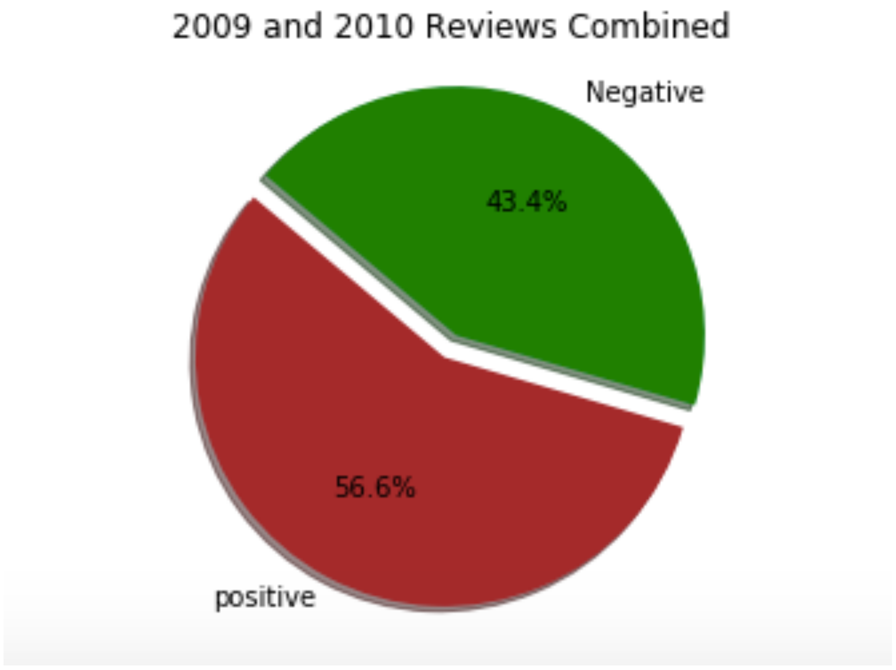
And In the combined section the overall

Positive:- 2899
Negative:- 2225

The stats for the above data:-




Reviews of 2010
Negative
44.4%
55.6%
positive


2009 and 2010 Reviews Combined
Negative
43.4%
56.6%
positive

After getting the sentences are as tagged positive and negative sentences. I used that sentences as my next corpus to train the Naiyes Classifier.

Classification:-

For Classification I tagged the sentences as 1 for positive and -1 for negative so It would be easier for implementing the label list and num_flods for cross validation.

I stored 5000+ sentences in CSV file and the sentiment for each sentences.

For each sentences I used 3 tokenizes:-

- Countvectorizer
- word_punct tokenizer
- word tokenizer

  - 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.

- 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.

I printed all the tokenizer part as first 10 sentences After seeing the results I used the word_tokenizer function as NLTK tokenize for further process.

- **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-

a. Word tokenizer without pre-processing (as discussed earlier)
b. 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 sentiment analysis.

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.

So here The final Pre-processed words for the list sentences are I combined in the final sentence in a function as :-

```
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
```

- **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.

```
#Filtering functions-
def rem_characters(doc):
  word_list=[]
  for (word,label) in doc:
    filtered_words = [x for x in word if len(x) > 2]
    word_list.extend(filtered_words)
  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
```

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 1 and -1 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')
  # convert each feature set to a line in the file with comma separated
feature values,
  # each feature value is converted to a string
  #   for booleans this is the words true and false
  #   for numbers, this is the string with the number
  for featureset in featuresets:
    featureline = ''
    for key in featurenames:
     try:
      featureline += str(featureset[0].get(key,[])) + ','
     except KeyError:
      continue
    if featureset[1] == -1:
     featureline += str("negative")
    elif featureset[1] == 1:
     featureline += str("positive")
    # write each feature set values to the file
    f.write(featureline)
    f.write('\n')
  f.close()
```

Feature Selection-

Bag of words feature Function:-

```
def bag_of_words(wordlist,wordcount):
#def bag_of_words(wordlist,wordcountCloud Storage: Centralized or
Decentralized- Architecture, Blockchain and Beyond):
  wordlist = nltk.FreqDist(wordlist)
  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 below.

Bag of words for bigram:
from nltk.collocations import *

```
def bag_of_words_bigram(wordlist,bigramcount):
  bigram_measures = nltk.collocations.BigramAssocMeasures()
  finder = BigramCollocationFinder.from_words(wordlist,window_size=3)
  finder.apply_freq_filter(3)
  bigramword_features = finder.nbest(bigram_measures.chi_sq, 3000)
  return bigramword_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:
                  features['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 in the bag of words set

For Example:-

```
Unigramsets_without_preprocessing -
({'contains(.)': True, 'contains(the)': True, 'contains(,)': False, 'contains(I)':
True, 'contains(and)': True, 'contains(a)': True, 'contains(to)': True,
'contains(is)': True, 'contains(it)': False, 'contains(of)': False, 'contains(for)':
True, 'contains(in)': False, 'contains(are)': True, 'contains(my)': False,
'contains(that)': True, 'contains(this)': False, 'contains(on)': False,
'contains(have)': False, 'contains(but)': True, 'contains(with)': False,
'contains(they)': False, 'contains(!)': False, 'contains(them)': True,
'contains(you)': False, "contains(n't)": True, 'contains(not)': False,
'contains(these)': True, 'contains(The)': True, 'contains(was)': True,
'contains(as)': False, "contains('s)": False, 'contains(so)': False,
'contains(very)': False, 'contains())': False, 'contains(be)': False, 'contains(()':
False,……..
```

Bigram feature:

```
def bigram_features(doc,word_features,bigramword_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 bigramword_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 –

Bigramsets_without_preprocessing -
({'contains(.)': True, 'contains(the)': True, 'contains(,)': False, 'contains(I)': True, 'contains(and)': True, 'contains(a)': True, 'contains(to)': True, 'contains(is)': True, 'contains(it)': False, 'contains(of)': False, 'contains(for)': True, 'contains(in)': False, 'contains(are)': True, 'contains(my)': False, 'contains(that)': True, 'contains(this)': False, 'contains(on)': False, 'contains(have)': False, 'contains(but)': True, 'contains(with)': False, 'contains(they)': False, 'contains(!)': False, 'contains(them)': True, 'contains(you)': False, "contains(n't)": True, 'contains(not)': False, 'contains(these)': True, 'contains(The)': True, 'contains(was)': True, 'contains(as)': False, "contains('s)": False, 'contains(so)': False, 'contains(very)': False, 'contains())': False, 'contains(be)': False, 'contains(()': 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','questio
nable','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','s
tormy','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','unw
ise','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
```

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

  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
```

Negativesets_without_preprocessing -

```
({'contains(.)': True, 'contains(NOT.)': False, 'contains(the)': True, 'contains(NOTthe)': False, 'contains(,)': False, 'contains(NOT,)': False, 'contains(I)': True, 'contains(NOTI)': False, 'contains(and)': True, 'contains(NOTand)': True, 'contains(a)': True, 'contains(NOTa)': False, 'contains(to)': True, 'contains(NOTto)': False, 'contains(is)': True, 'contains(NOTis)': False, 'contains(it)': False, 'contains(NOTit)': False, 'contains(of)': False, 'contains(NOTof)': False, 'contains(for)': True, 'contains(NOTfor)': False, 'contains(in)': False, 'contains(NOTin)': False, 'contains(are)': True, 'contains(NOTare)': False, 'contains(my)': 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(.)': True, 'contains(the)': True, 'contains(,)': False, 'contains(I)': True, 'contains(and)': True, 'contains(a)': True, 'contains(to)': True, 'contains(is)': True, 'contains(it)': False, 'contains(of)': False, 'contains(for)': True, 'contains(in)': False, 'contains(are)': True, 'contains(my)': False, 'contains(that)': True, 'contains(this)': False, 'contains(on)': False, 'contains(have)': False,……
'contains(inside)': False, 'contains(thin)': False, 'contains(thought)': False, 'contains(easily)': False, 'contains(buying)': False, 'contains(sandals)': False, 'nouns': 13, 'verbs': 11, 'adjectives': 3, 'adverbs': 5}, -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')
        # initialize an empty dictionary
        sldict = { }
        for line in flexicon:
                fields = line.split()   # default is to split on whitespace
                # split each field on the '=' and keep the second part as the
value
                strength = fields[0].split("=")[1]
                word = fields[2].split("=")[1]
                posTag = fields[3].split("=")[1]
                stemmed = fields[4].split("=")[1]
                polarity = fields[5].split("=")[1]
                if (stemmed == 'y'):
                        isStemmed = True
                else:
                        isStemmed = False
                # put a dictionary entry with the word as the keyword
                #     and a list of the other values
                sldict[word] = [strength, posTag, isStemmed, polarity]
        return sldict
```

```
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.

I used the words from this function as it was giving the output as desired,

```
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

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

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
```

For example- -

Subjectivitysets_without_preprocessing -
({'contains(.)': True, 'contains(the)': True, 'contains(,)': False, 'contains(I)': True, 'contains(and)': True, 'contains(a)': True, 'contains(to)': True, 'contains(is)': True, 'contains(it)': False, 'contains(of)': False,……. 'positivecount': 7, 'negativecount': 1}, 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
```

liwcsets_without_preprocessing -
({'contains(.)': True, 'contains(the)': True, 'contains(,)': False, 'contains(I)': True, 'contains(and)': True, 'contains(a)': True, 'contains(to)': True, 'contains(is)': True, 'contains(it)': False, 'contains(of)': False,…… 'positivecount': 3, 'negativecount': 0}, -1)

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
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 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('corpus','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',encoding='latin-
1').readlines()[35:]:
    pos_words = re.findall(r"[\w']+|[.,!?;]", line.rstrip())
    pos_features.append(pos_words[0])

  for line in open(NEGATIVE_REVIEWS, 'r',encoding='latin-
1').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. It's the same as the movie_review corpus.

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,"corpus/outputcsv/un
igramsets_without_preprocessing.csv")
  print(" ")
```

Whatever I am getting the featureset I am saving that on CSV file to store for future purpose and record.

NLTK Classifiers-

Naive Bayes Classifier:

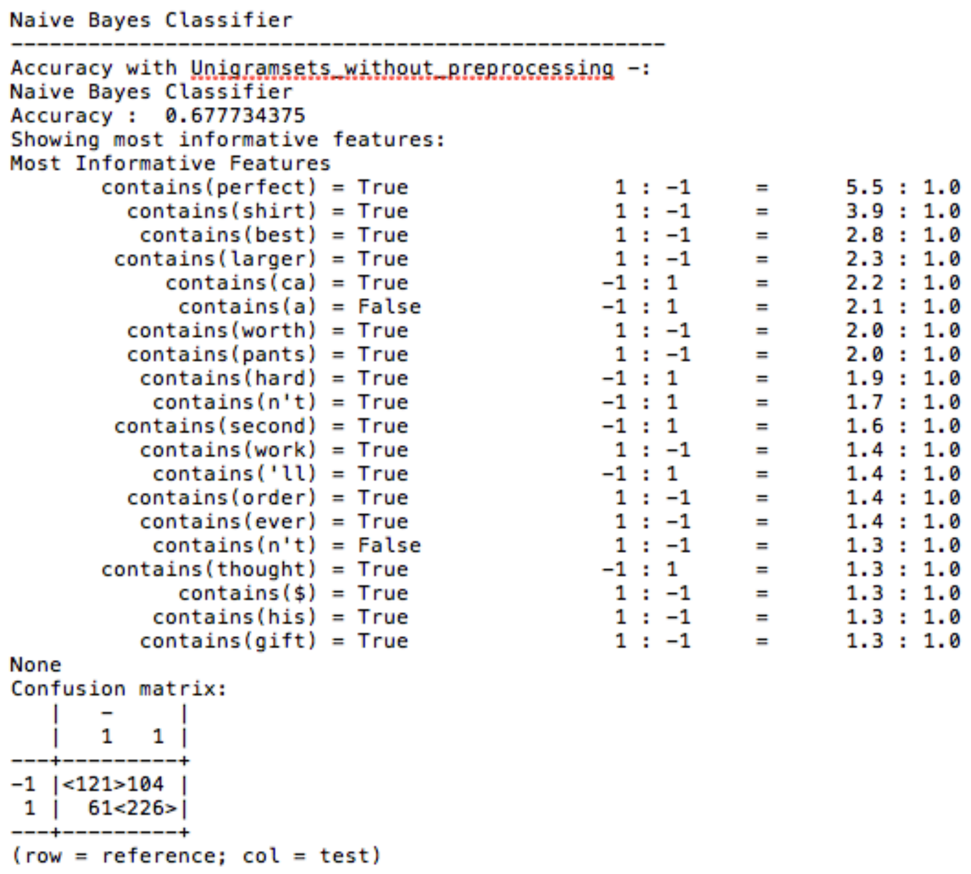
I am using Naïve Bayes classifier to train and test data with 90 % of data as training set and 10% as test set initially.

```
#Classifer functions(Single-fold)
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(20))
  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, For prediction the function is as follows:-

```
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:-



```
Naive Bayes Classifier
---------------------------------------------------
Accuracy with Unigramsets_without_preprocessing -:
Naive Bayes Classifier
Accuracy :  0.677734375
Showing most informative features:
Most Informative Features
       contains(perfect) = True              1 : -1     =      5.5 : 1.0
         contains(shirt) = True              1 : -1     =      3.9 : 1.0
          contains(best) = True              1 : -1     =      2.8 : 1.0
        contains(larger) = True              1 : -1     =      2.3 : 1.0
            contains(ca) = True             -1 : 1      =      2.2 : 1.0
             contains(a) = False            -1 : 1      =      2.1 : 1.0
         contains(worth) = True              1 : -1     =      2.0 : 1.0
         contains(pants) = True              1 : -1     =      2.0 : 1.0
          contains(hard) = True             -1 : 1      =      1.9 : 1.0
           contains(n't) = True             -1 : 1      =      1.7 : 1.0
        contains(second) = True             -1 : 1      =      1.6 : 1.0
          contains(work) = True              1 : -1     =      1.4 : 1.0
           contains('ll) = True             -1 : 1      =      1.4 : 1.0
         contains(order) = True              1 : -1     =      1.4 : 1.0
          contains(ever) = True              1 : -1     =      1.4 : 1.0
           contains(n't) = False             1 : -1     =      1.3 : 1.0
       contains(thought) = True             -1 : 1      =      1.3 : 1.0
             contains($) = True              1 : -1     =      1.3 : 1.0
           contains(his) = True              1 : -1     =      1.3 : 1.0
          contains(gift) = True              1 : -1     =      1.3 : 1.0
None
Confusion matrix:
   |   -     |
   |   1   1 |
---+---------+
-1 |<121>104 |
 1 |  61<226>|
---+---------+
(row = reference; col = test)
```

Maximum Entropy Classifier-

Max Entropy classifier is a probabilistic classifier which belongs to the class of exponential models. Unlike the Naive Bayes 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.

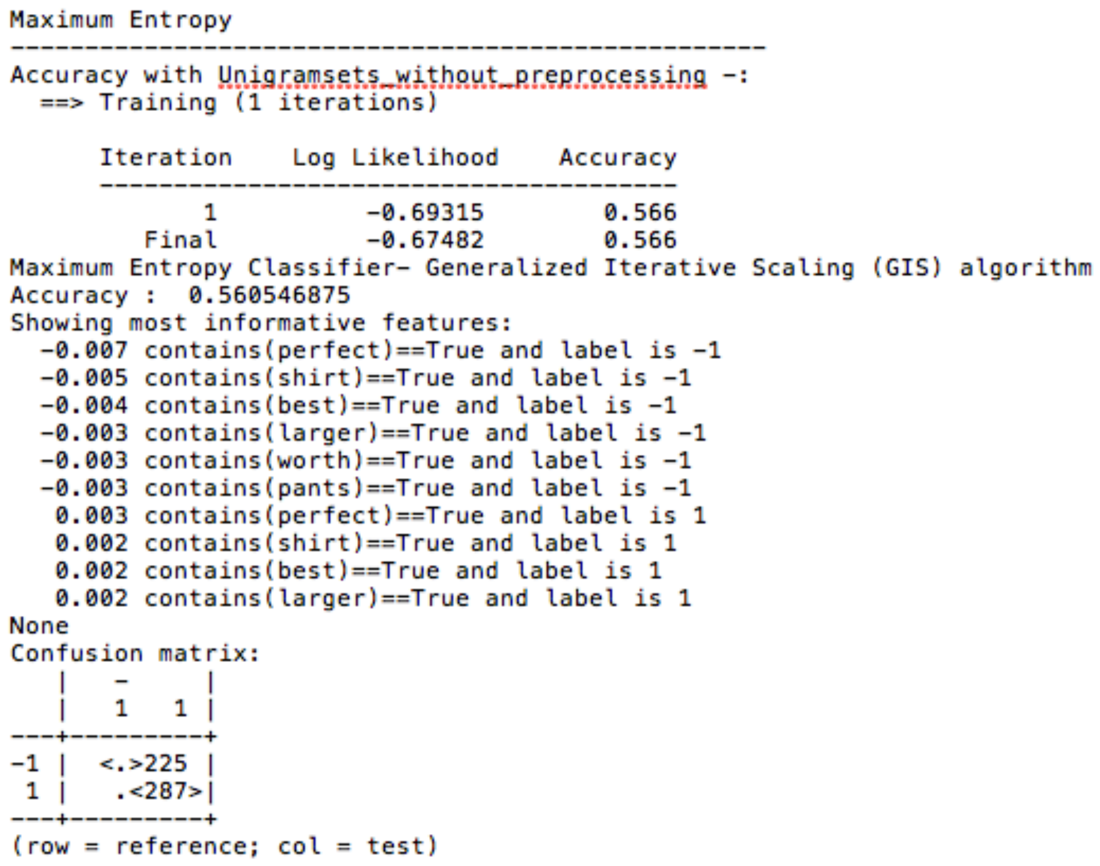
We are going to use three different algorithms of max entropy to train and test our data:

a.) Generalized Iterative Scaling (GIS) algorithm

b.) Improved Iterative Scaling (IIS)

```
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:-



```
Maximum Entropy
--------------------------------------------------
Accuracy with Unigramsets_without_preprocessing -:
  ==> Training (1 iterations)

      Iteration    Log Likelihood    Accuracy
      ---------------------------------------
             1          -0.69315        0.566
         Final          -0.67482        0.566
Maximum Entropy Classifier- Generalized Iterative Scaling (GIS) algorithm
Accuracy :  0.560546875
Showing most informative features:
  -0.007 contains(perfect)==True and label is -1
  -0.005 contains(shirt)==True and label is -1
  -0.004 contains(best)==True and label is -1
  -0.003 contains(larger)==True and label is -1
  -0.003 contains(worth)==True and label is -1
  -0.003 contains(pants)==True and label is -1
   0.003 contains(perfect)==True and label is 1
   0.002 contains(shirt)==True and label is 1
   0.002 contains(best)==True and label is 1
   0.002 contains(larger)==True and label is 1
None
Confusion matrix:
   |   -     |
   |   1   1 |
---+---------+
-1 |  <.>225 |
 1 |   .<287>|
---+---------+
(row = reference; col = test)
```

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

- Logistic Regression
- Random Forest
- MultinomiaINB
- BernoulliNB
- SVC
- Linear SVC
- NUSVC
- SGDClassifier
- Decision Tree

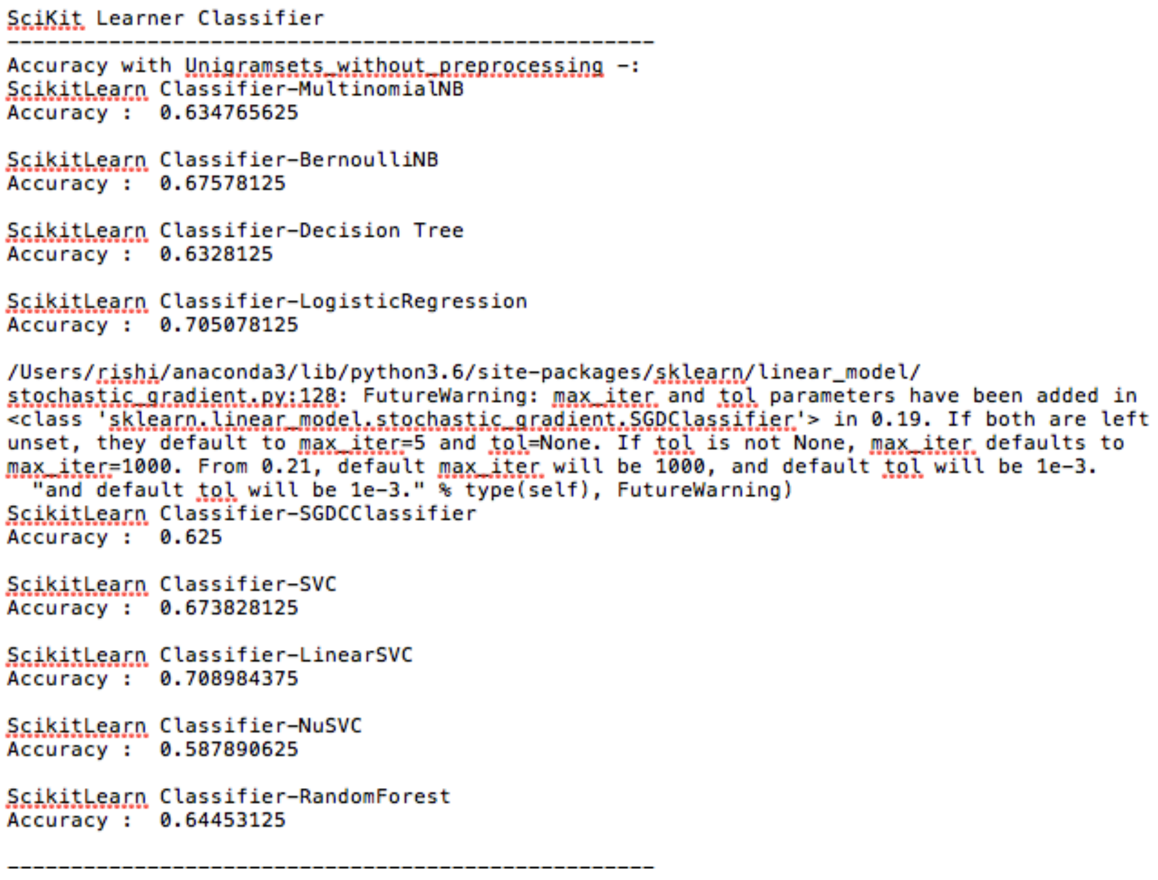
The code to implement these algorithms:-

```
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

ef 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=0.01))
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:-



```
SciKit Learner Classifier
---------------------------------------------------
Accuracy with Unigramsets_without_preprocessing -:
ScikitLearn Classifier-MultinomialNB
Accuracy :  0.634765625

ScikitLearn Classifier-BernoulliNB
Accuracy :  0.67578125

ScikitLearn Classifier-Decision Tree
Accuracy :  0.6328125

ScikitLearn Classifier-LogisticRegression
Accuracy :  0.705078125

/Users/rishi/anaconda3/lib/python3.6/site-packages/sklearn/linear_model/
stochastic_gradient.py:128: FutureWarning: max_iter and tol parameters have been added in
<class 'sklearn.linear_model.stochastic_gradient.SGDClassifier'> in 0.19. If both are left
unset, they default to max_iter=5 and tol=None. If tol is not None, max_iter defaults to
max_iter=1000. From 0.21, default max_iter will be 1000, and default tol will be 1e-3.
  "and default tol will be 1e-3." % type(self), FutureWarning)
ScikitLearn Classifier-SGDClassifier
Accuracy :  0.625

ScikitLearn Classifier-SVC
Accuracy :  0.673828125

ScikitLearn Classifier-LinearSVC
Accuracy :  0.708984375

ScikitLearn Classifier-NuSVC
Accuracy :  0.587890625

ScikitLearn Classifier-RandomForest
Accuracy :  0.64453125

---------------------------------------------------
```

Single Fold Performances of all classifiers against all feature sets –

With Preprocessing

	Naïve Bayes	GIS	IIS	Multi Nomial NB	Bernoulli NB	Decision Tree	Logistic Regression	SGDC	SVC	Linear SVC	Nu SVC	Random Forest	AVG
Unigram	0.654	0.560	0.560	0.660	0.6562	0.5937	0.6582	0.625	0.609	0.666	0.5976	0.6191	**0.6215**
Bigram	0.654	0.560	0.439	0.644	0.6425	0.6152	0.660	0.603	0.5625	0.666	0.533	0.632	**0.6009**
POS	0.6542	0.566	0.437	0.628	0.636	0.574	0.6523	0.560	0.599	0.658	0.570	0.601	**0.594625**
Negation	0.665	0.560	0.560	0.591	0.570	0.5859	0.6328	0.6386	0.560	0.6347	0.5351	0.6171	**0.59585**
SL	0.6582	0.560	0.560	0.630	0.652	0.619	0.671	0.611	0.5839	0.6582	0.5703	0.6445	**0.618175**
LIWC	0.6503	0.560	0.560	0.6347	0.6386	0.601	0.666	0.613	0.6191	0.6582	0.5820	0.623	**0.617158333**
SL+ LIWC	0.6406	0.560	0.560	0.640	0.652	0.582	0.667	0.669	0.585	0.666	0.541	0.5839	**0.612208333**
Opinion	0.6699	0.560	0.560	0.6347	0.6386	0.5957	0.6660	0.6308	0.6191	0.6718	0.5820	0.6054	**0.6195**
AvG :-	**0.655775**	**0.56075**	**0.5295**	**0.6328**	**0.6357375**	**0.5958125**	**0.6591625**	**0.6188**	**0.5922**	**0.6598625**	**0.563875**	**0.61575**	

Without Preprocessing:-

	Naïve Bayes	GIS	IIS	Multi Nomial NB	Bernoulli NB	Decision Tree	Logistic Regression	SGDC	SVC	Linear SVC	Nu SVC	Random Forest	Avg.
Unigram	0.677	0.560	0.560	0.634	0.675	0.632	0.705	0.625	0.6738	0.7089	0.5878	0.6445	**0.64025**
Bigram	0.677	0.560	0.503	0.630	0.667	0.605	0.705	0.671	0.623	0.707	0.5449	0.6074	**0.625025**
POS	0.679	0.560	0.439	0.5625	0.55664	0.5917	0.7011	0.6152	0.5605	0.6953	0.5332	0.5546	**0.587395**
Negation	0.673	0.560	0.439	0.630	0.675	0.585	0.705	0.6425	0.667	0.6777	0.6269	0.6621	**0.6286**
SL	0.669	0.560	0.560	0.6230	0.6601	0.6464	0.6953	0.6308	0.6464	0.6894	0.6152	0.6621	**0.63814**
LIWC	0.666	0.560	0.560	0.6230	0.6777	0.5878	0.6972	0.4921	0.6621	0.667	0.615	0.6621	**0.6225**
SL+ LIWC	0.666	0.560	0.560	0.6230	0.677	0.5878	0.6972	0.4921	0.6621	0.6621	0.6269	0.6582	**0.6227**
Opinion	0.669	0.560	0.560	0.6230	0.660	0.626	0.695	0.572	0.646	0.6914	0.6152	0.6386	**0.62968**
Average:-	**0.672**	**0.56**	**0.522625**	**0.6185625**	**0.656055**	**0.6077125**	**0.7001**	**0.5925875**	**0.6426125**	**0.68735**	**0.5956375**	**0.6362**	

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, Note:- This has taken the longest time to execute.

```
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):
        try:
            ref_sets[label].add(j)
        except KeyError:
            print("KeyError encountered")
            #print(label)
    # get predicted labels
    for k, label in enumerate(testlist):
        try:
            test_sets[label].add(k)
        except KeyError:
            print("KeyError encountered")
            #print(label)

    # 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)
```

The formal definition of Precision, Recall and F Score

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)

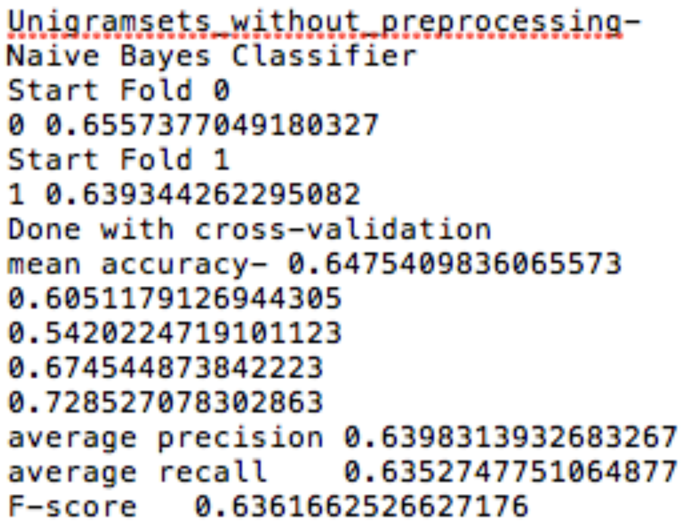
Recall = TP/TP+FN

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. I achieved similar levels of F1- measure (0.63)

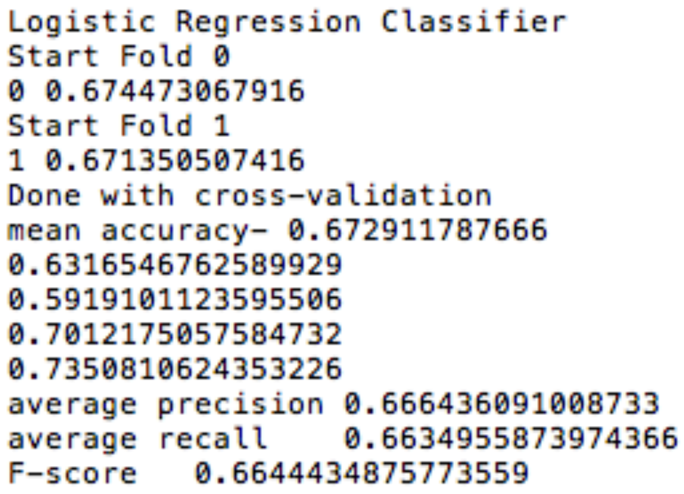
F1 Score = 2*(Recall * Precision) / (Recall + Precision)

For Cross Validation It has generated lots of data and accuracy according to respective algorithms. Here I am presenting few screenshots for the Grader’s Understanding.(The best one’s) [Note:-I fetch some code from the internet and modified it to out requirements:]-

For Unigrams Without Processing:-

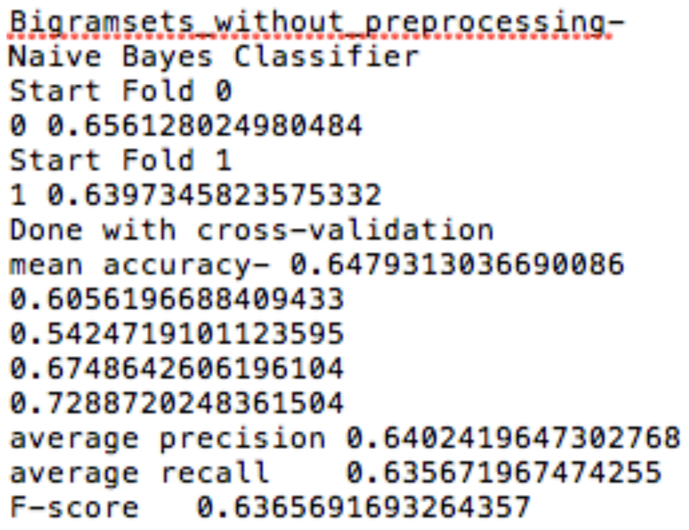


```
Unigramsets_without_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.6557377049180327
Start Fold 1
1 0.639344262295082
Done with cross-validation
mean accuracy- 0.6475409836065573
0.6051179126944305
0.5420224719101123
0.674544873842223
0.728527078302863
average precision 0.6398313932683267
average recall    0.6352747751064877
F-score   0.6361662526627176
```

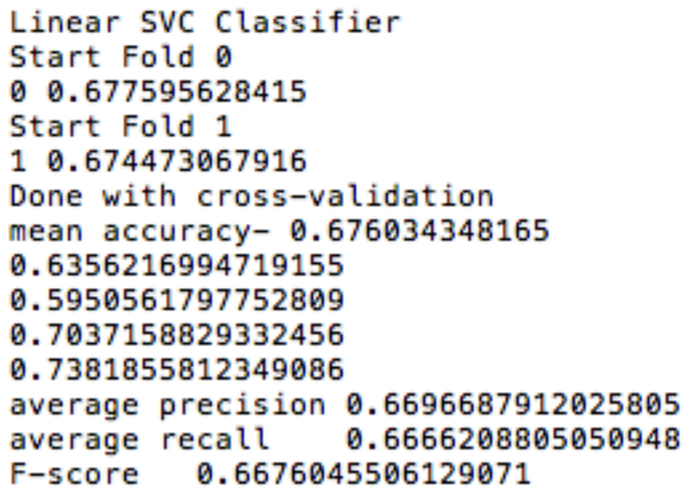


```
Logistic Regression Classifier
Start Fold 0
0 0.674473067916
Start Fold 1
1 0.671350507416
Done with cross-validation
mean accuracy- 0.672911787666
0.6316546762589929
0.5919101123595506
0.7012175057584732
0.7350810624353226
average precision 0.666436091008733
average recall    0.6634955873974366
F-score   0.6644434875773559
```

For Bigrams Without Processing:-

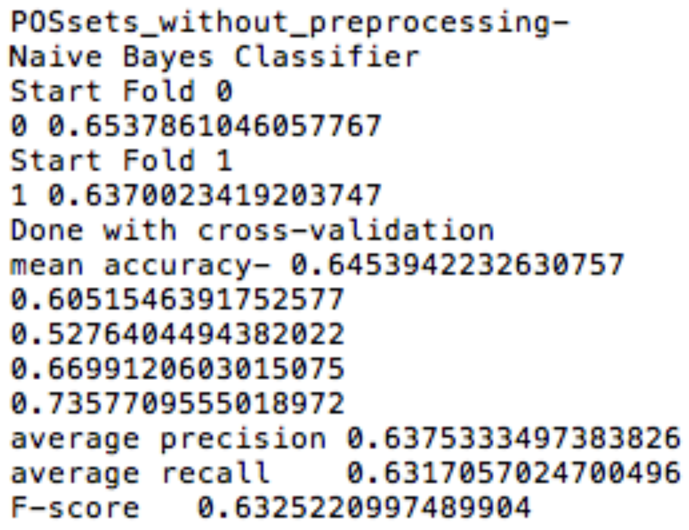


```
Bigramsets_without_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.656128024980484
Start Fold 1
1 0.6397345823575332
Done with cross-validation
mean accuracy- 0.6479313036690086
0.6056196688409433
0.5424719101123595
0.6748642606196104
0.7288720248361504
average precision 0.6402419647302768
average recall    0.635671967474255
F-score   0.6365691693264357
```

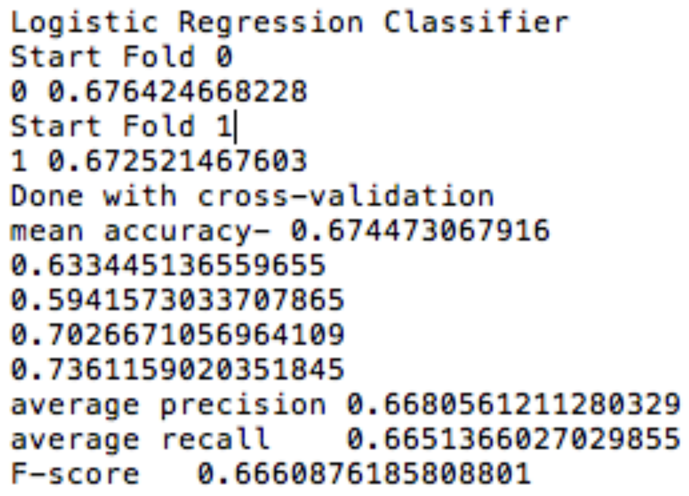


```
Linear SVC Classifier
Start Fold 0
0 0.677595628415
Start Fold 1
1 0.674473067916
Done with cross-validation
mean accuracy- 0.676034348165
0.6356216994719155
0.5950561797752809
0.7037158829332456
0.7381855812349086
average precision 0.6696687912025805
average recall    0.6666208805050948
F-score   0.6676045506129071
```

For POSSet without preprocessing:-

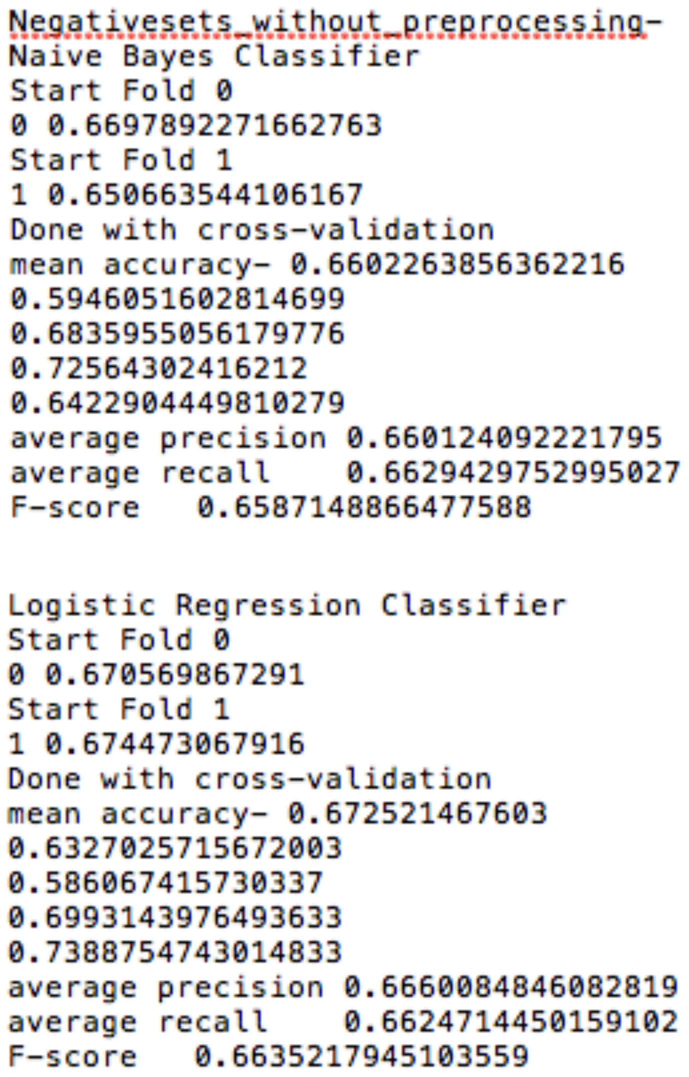


```
POSsets_without_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.6537861046057767
Start Fold 1
1 0.6370023419203747
Done with cross-validation
mean accuracy- 0.6453942232630757
0.6051546391752577
0.5276404494382022
0.6699120603015075
0.7357709555018972
average precision 0.6375333497383826
average recall    0.6317057024700496
F-score   0.6325220997489904
```



```
Logistic Regression Classifier
Start Fold 0
0 0.676424668228
Start Fold 1
1 0.672521467603
Done with cross-validation
mean accuracy- 0.674473067916
0.633445136559655
0.5941573033707865
0.7026671056964109
0.7361159020351845
average precision 0.6680561211280329
average recall    0.6651366027029855
F-score   0.6660876185808801
```

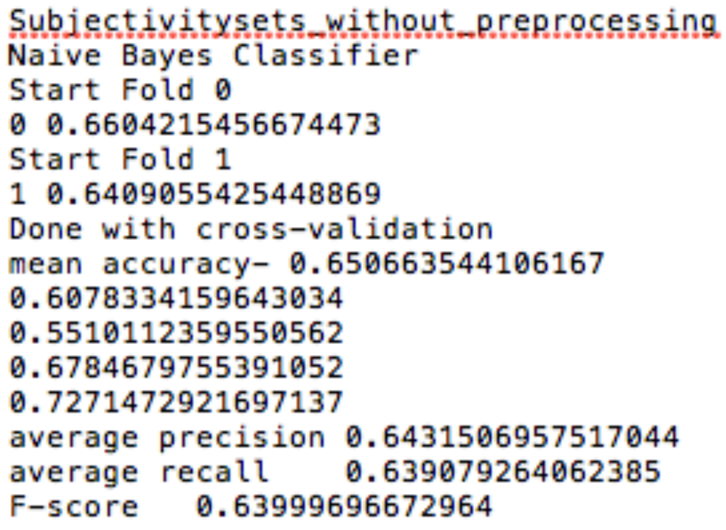
For NegativeSet without Preprocessing:-



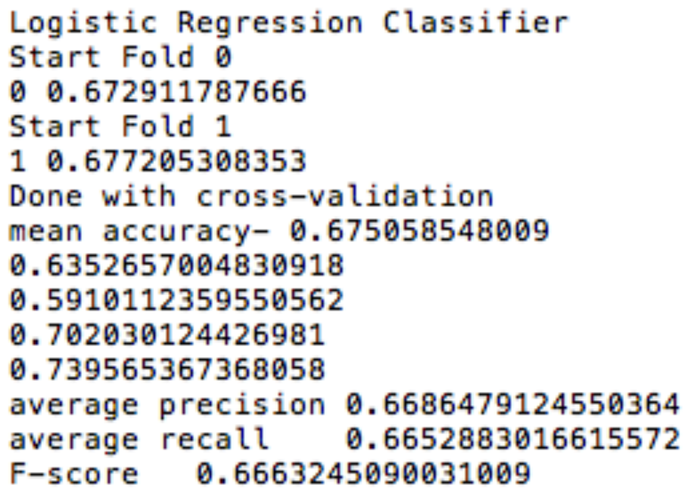
```
Negativesets_without_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.6697892271662763
Start Fold 1
1 0.650663544106167
Done with cross-validation
mean accuracy- 0.6602263856362216
0.5946051602814699
0.6835955056179776
0.72564302416212
0.6422904449810279
average precision 0.660124092221795
average recall    0.6629429752995027
F-score   0.6587148866477588


Logistic Regression Classifier
Start Fold 0
0 0.670569867291
Start Fold 1
1 0.674473067916
Done with cross-validation
mean accuracy- 0.672521467603
0.6327025715672003
0.586067415730337
0.6993143976493633
0.7388754743014833
average precision 0.6660084846082819
average recall    0.6624714450159102
F-score   0.6635217945103559
```

Subjectivityset without preprocessing:-

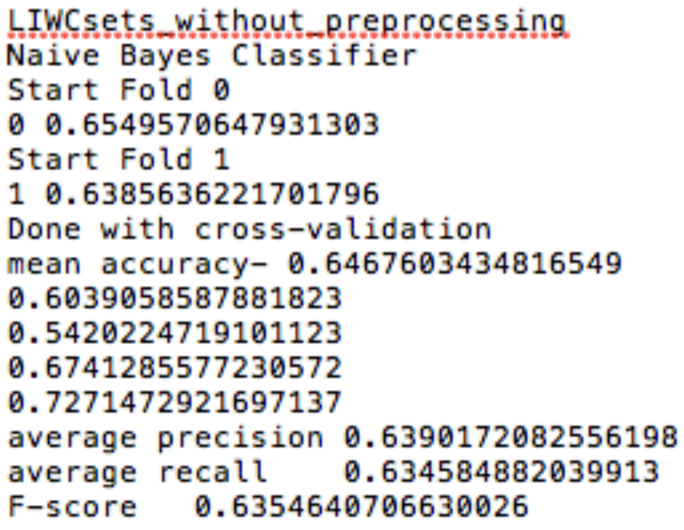


```
Subjectivitysets_without_preprocessing
Naive Bayes Classifier
Start Fold 0
0 0.6604215456674473
Start Fold 1
1 0.6409055425448869
Done with cross-validation
mean accuracy- 0.650663544106167
0.6078334159643034
0.5510112359550562
0.6784679755391052
0.7271472921697137
average precision 0.6431506957517044
average recall    0.639079264062385
F-score   0.63999696672964
```

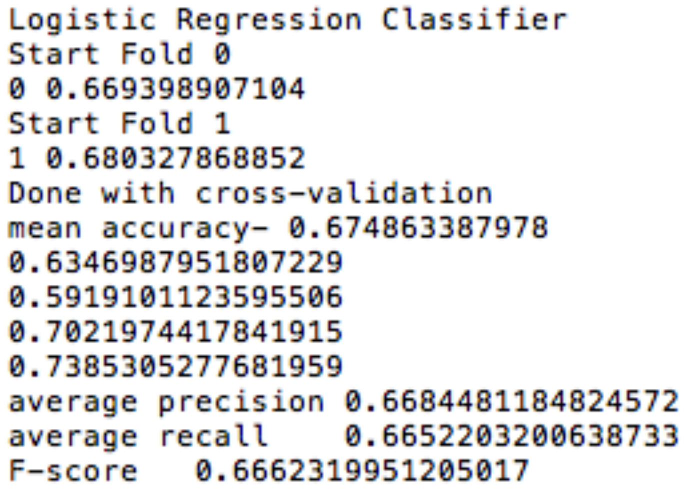


```
Logistic Regression Classifier
Start Fold 0
0 0.672911787666
Start Fold 1
1 0.677205308353
Done with cross-validation
mean accuracy- 0.675058548009
0.6352657004830918
0.5910112359550562
0.702030124426981
0.739565367368058
average precision 0.6686479124550364
average recall    0.6652883016615572
F-score   0.6663245090031009
```

LIWCsets without preprocessing:-

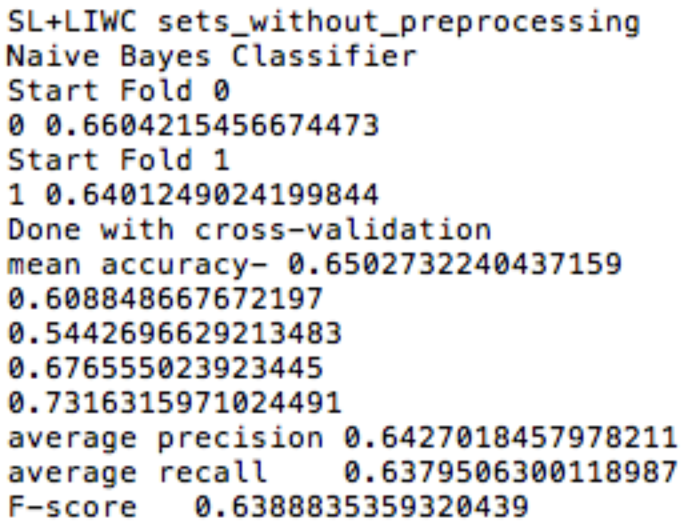


```
LIWCsets_without_preprocessing
Naive Bayes Classifier
Start Fold 0
0 0.6549570647931303
Start Fold 1
1 0.6385636221701796
Done with cross-validation
mean accuracy- 0.6467603434816549
0.6039058587881823
0.5420224719101123
0.6741285577230572
0.7271472921697137
average precision 0.6390172082556198
average recall    0.634584882039913
F-score   0.6354640706630026
```

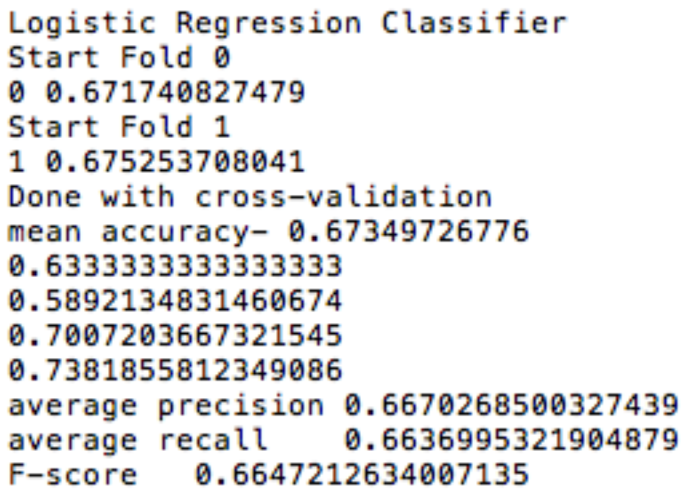


```
Logistic Regression Classifier
Start Fold 0
0 0.669398907104
Start Fold 1
1 0.680327868852
Done with cross-validation
mean accuracy- 0.674863387978
0.6346987951807229
0.5919101123595506
0.7021974417841915
0.7385305277681959
average precision 0.6684481184824572
average recall    0.6652203200638733
F-score   0.6662319951205017
```

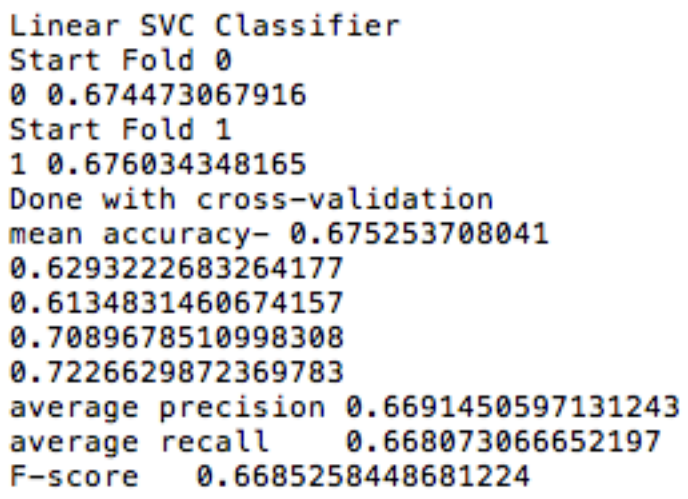
SL+LIWC Sets without preprocessing:-



```
SL+LIWC sets_without_preprocessing
Naive Bayes Classifier
Start Fold 0
0 0.6604215456674473
Start Fold 1
1 0.6401249024199844
Done with cross-validation
mean accuracy- 0.6502732240437159
0.608848667672197
0.5442696629213483
0.676555023923445
0.7316315971024491
average precision 0.6427018457978211
average recall    0.6379506300118987
F-score   0.6388835359320439
```

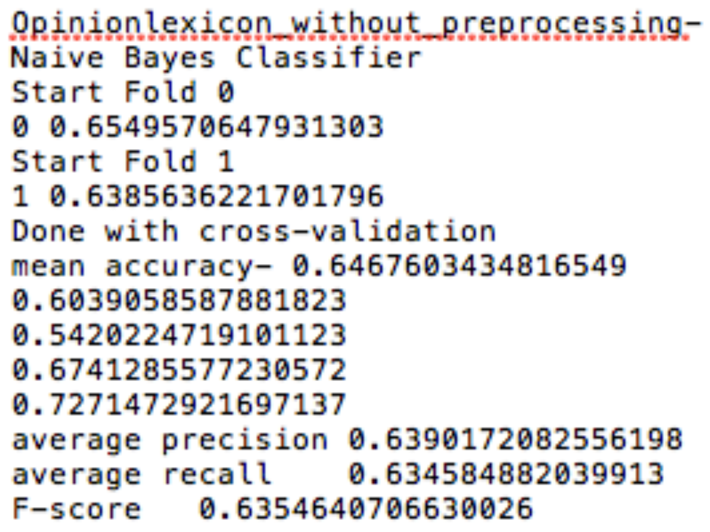


```
Logistic Regression Classifier
Start Fold 0
0 0.671740827479
Start Fold 1
1 0.675253708041
Done with cross-validation
mean accuracy- 0.67349726776
0.6333333333333333
0.5892134831460674
0.7007203667321545
0.7381855812349086
average precision 0.6670268500327439
average recall    0.6636995321904879
F-score   0.6647212634007135
```

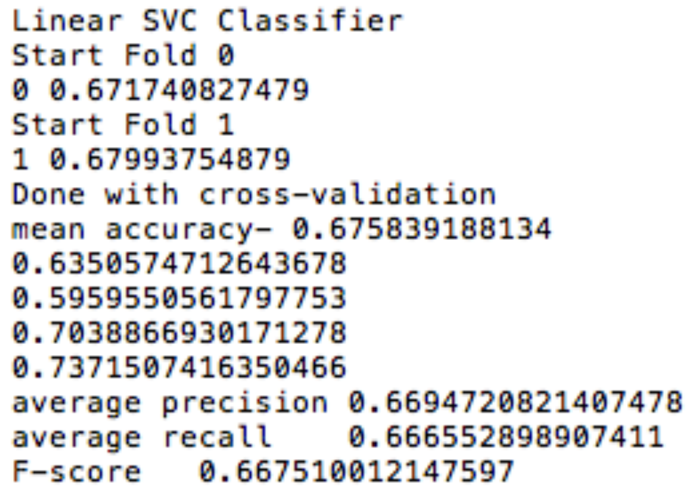


```
Linear SVC Classifier
Start Fold 0
0 0.674473067916
Start Fold 1
1 0.676034348165
Done with cross-validation
mean accuracy- 0.675253708041
0.629322683264177
0.613483146067415 7
0.7089678510998308
0.7226629872369783
average precision 0.6691450597131243
average recall    0.668073066652197
F-score   0.6685258448681224
```

Opinionlexicon without preprocessing:-



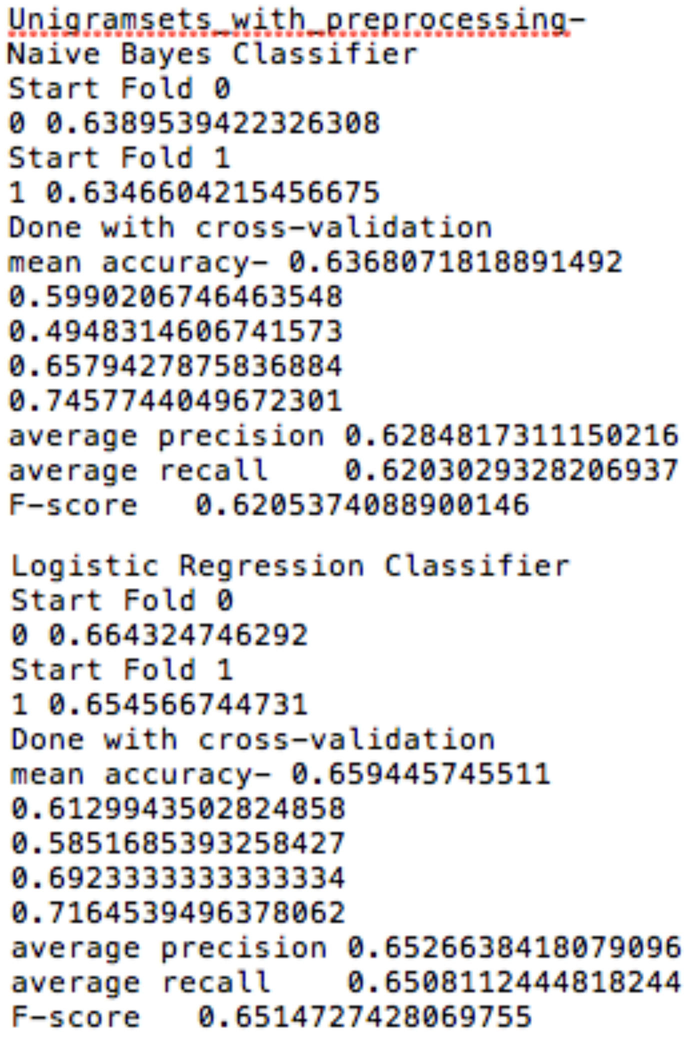
```
Opinionlexicon_without_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.6549570647931303
Start Fold 1
1 0.6385636221701796
Done with cross-validation
mean accuracy- 0.6467603434816549
0.6039058587881823
0.5420224719101123
0.6741285577230572
0.7271472921697137
average precision 0.6390172082556198
average recall    0.634584882039913
F-score   0.6354640706630026
```



```
Linear SVC Classifier
Start Fold 0
0 0.671740827479
Start Fold 1
1 0.67993754879
Done with cross-validation
mean accuracy- 0.675839188134
0.6350574712643678
0.5959550561797753
0.7038866930171278
0.7371507416350466
average precision 0.6694720821407478
average recall    0.666552898907411
F-score   0.667510012147597
```

Preprocessing Data:-

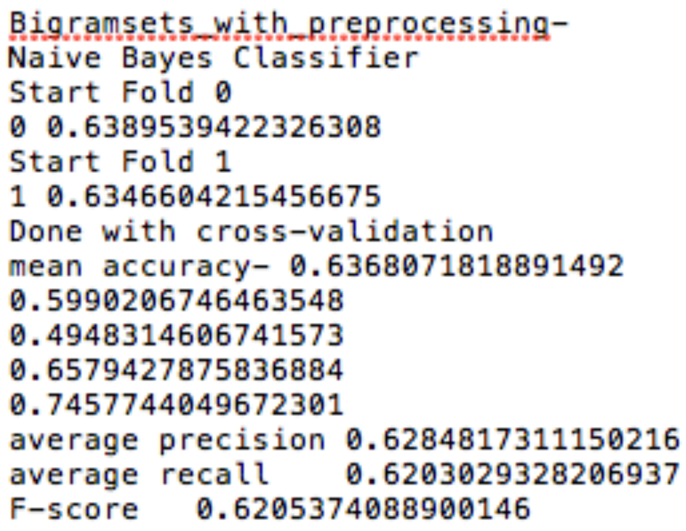
Unigrams with Pre-Processing:-



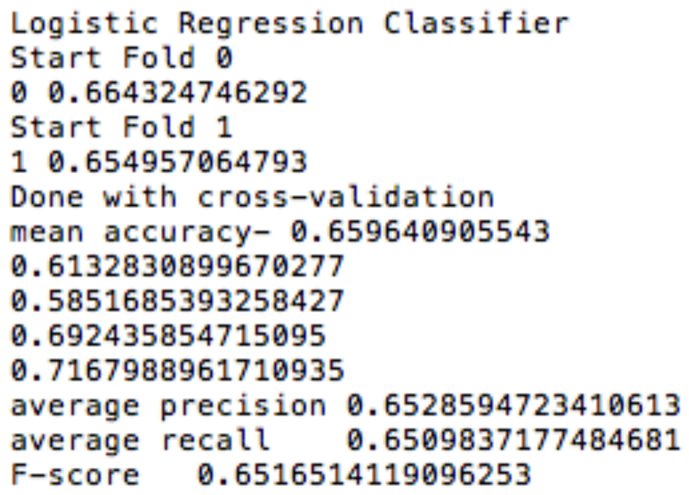
```
Unigramsets_with_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.6389539422326308
Start Fold 1
1 0.6346604215456675
Done with cross-validation
mean accuracy- 0.6368071818891492
0.5990206746463548
0.4948314606741573
0.6579427875836884
0.7457744049672301
average precision 0.6284817311150216
average recall    0.6203029328206937
F-score   0.6205374088900146

Logistic Regression Classifier
Start Fold 0
0 0.664324746292
Start Fold 1
1 0.654566744731
Done with cross-validation
mean accuracy- 0.659445745511
0.6129943502824858
0.5851685393258427
0.6923333333333334
0.7164539496378062
average precision 0.6526638418079096
average recall    0.6508112444818244
F-score   0.6514727428069755
```

Bigramsets with Preprocessing:-

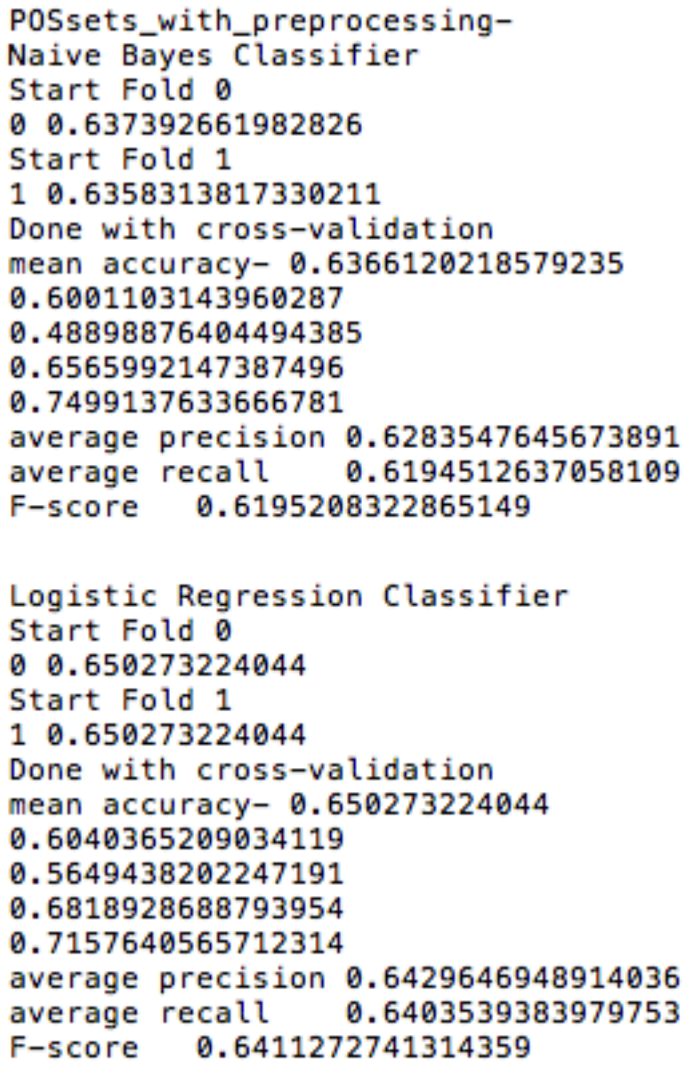


```
Bigramsets_with_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.6389539422326308
Start Fold 1
1 0.6346604215456675
Done with cross-validation
mean accuracy- 0.6368071818891492
0.5990206746463548
0.4948314606741573
0.6579427875836884
0.7457744049672301
average precision 0.6284817311150216
average recall    0.6203029328206937
F-score   0.6205374088900146
```



```
Logistic Regression Classifier
Start Fold 0
0 0.664324746292
Start Fold 1
1 0.654957064793
Done with cross-validation
mean accuracy- 0.659640905543
0.6132830899670277
0.5851685393258427
0.692435854715095
0.7167988961710935
average precision 0.6528594723410613
average recall    0.6509837177484681
F-score   0.6516514119096253
```

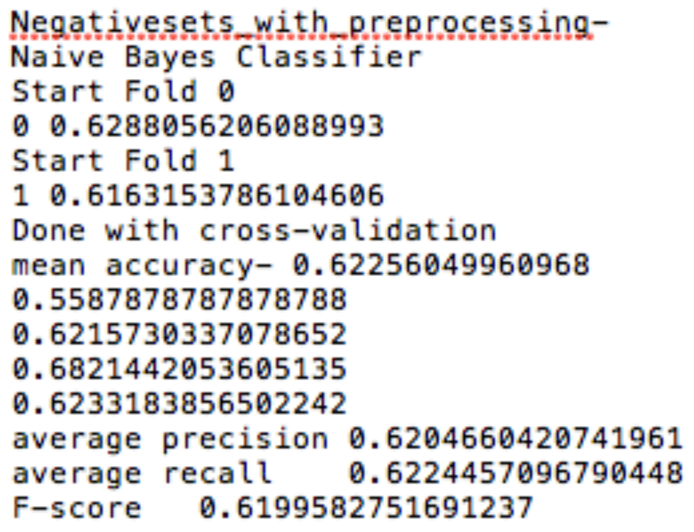
POSsets with Preprocessing



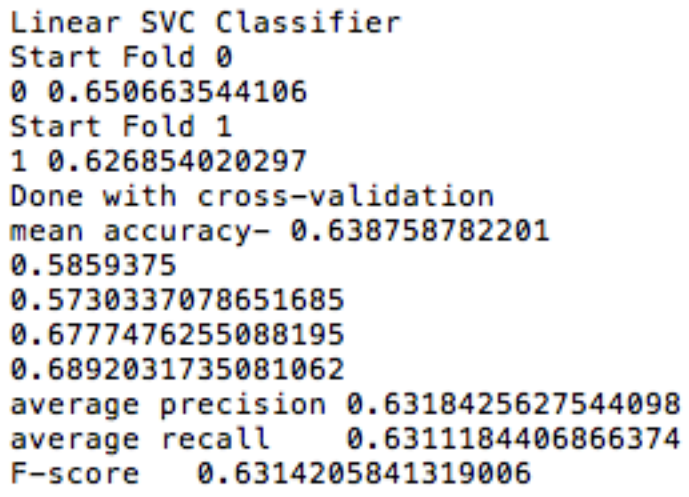
```
POSsets_with_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.637392661982826
Start Fold 1
1 0.6358313817330211
Done with cross-validation
mean accuracy- 0.6366120218579235
0.6001103143960287
0.48898876404494385
0.6565992147387496
0.7499137633666781
average precision 0.6283547645673891
average recall    0.6194512637058109
F-score   0.6195208322865149

Logistic Regression Classifier
Start Fold 0
0 0.650273224044
Start Fold 1
1 0.650273224044
Done with cross-validation
mean accuracy- 0.650273224044
0.6040365209034119
0.5649438202247191
0.6818928688793954
0.7157640565712314
average precision 0.6429646948914036
average recall    0.6403539383979753
F-score   0.6411272741314359
```

Negativesets with Preprocessing:-

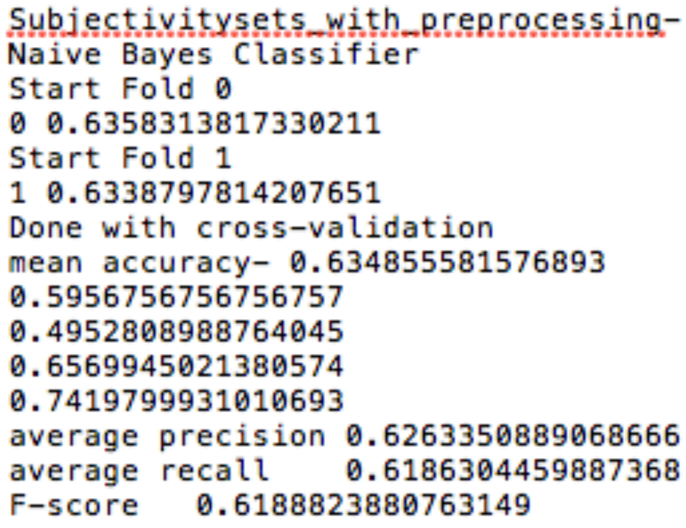


```
Negativesets_with_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.6288056206088993
Start Fold 1
1 0.6163153786104606
Done with cross-validation
mean accuracy- 0.62256049960968
0.5587878787878788
0.6215730337078652
0.6821442053605135
0.6233183856502242
average precision 0.6204660420741961
average recall    0.6224457096790448
F-score   0.6199582751691237
```

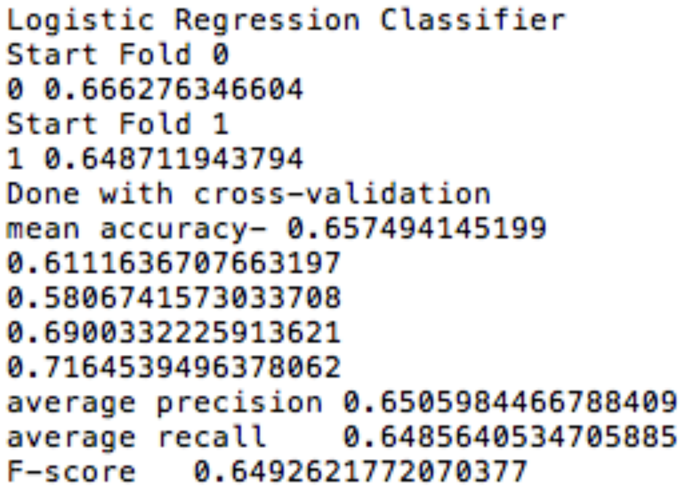


```
Linear SVC Classifier
Start Fold 0
0 0.650663544106
Start Fold 1
1 0.626854020297
Done with cross-validation
mean accuracy- 0.638758782201
0.5859375
0.5730337078651685
0.6777476255088195
0.6892031735081062
average precision 0.6318425627544098
average recall    0.6311184406866374
F-score   0.6314205841319006
```

Subjectivitusets with Preprocessing:-

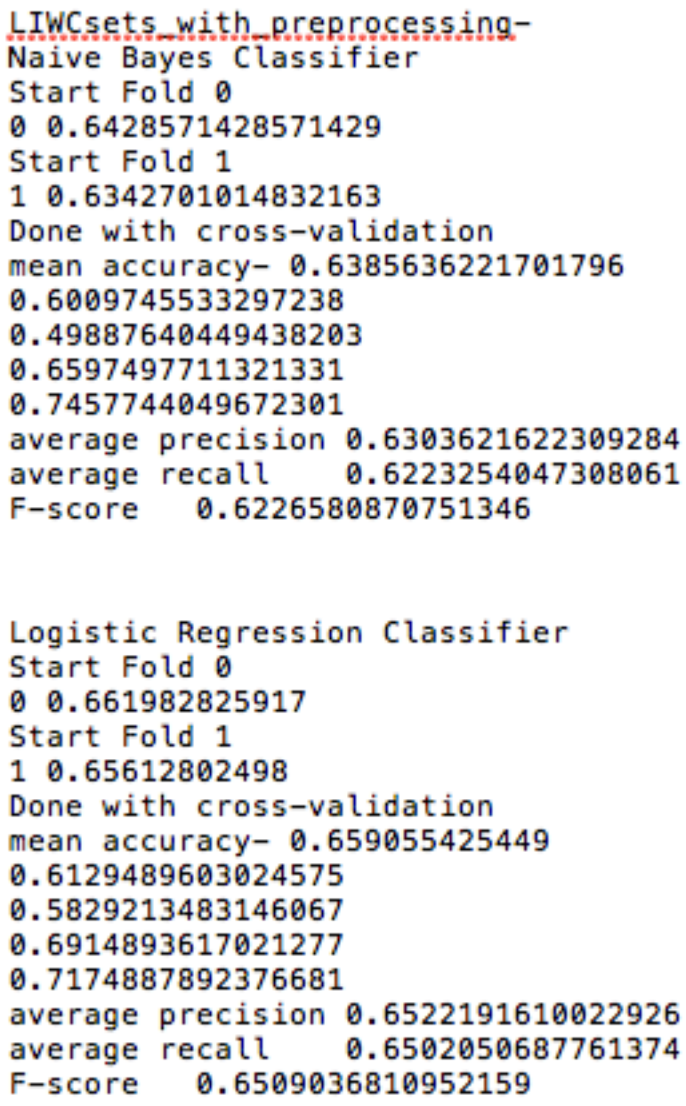


```
Subjectivitysets_with_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.6358313817330211
Start Fold 1
1 0.6338797814207651
Done with cross-validation
mean accuracy- 0.634855581576893
0.5956756756756757
0.4952808988764045
0.6569945021380574
0.7419799931010693
average precision 0.6263350889068666
average recall    0.6186304459887368
F-score   0.6188823880763149
```



```
Logistic Regression Classifier
Start Fold 0
0 0.666276346604
Start Fold 1
1 0.648711943794
Done with cross-validation
mean accuracy- 0.657494145199
0.6111636707663197
0.5806741573033708
0.6900332225913621
0.7164539496378062
average precision 0.6505984466788409
average recall    0.6485640534705885
F-score   0.6492621772070377
```

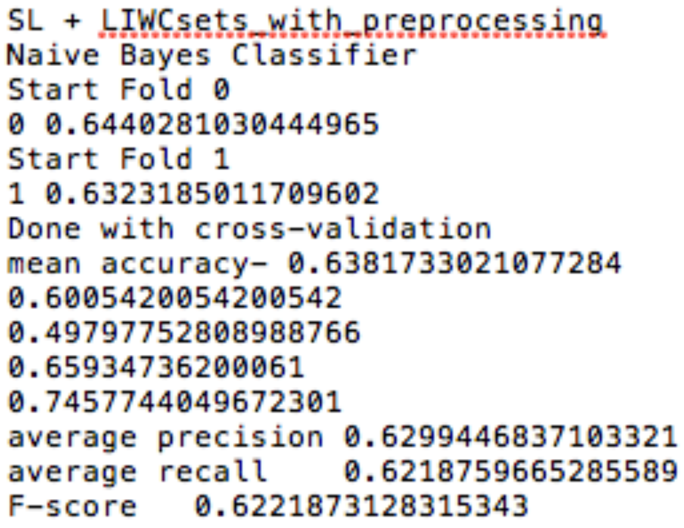
LIWCsets with Preprocessing:-



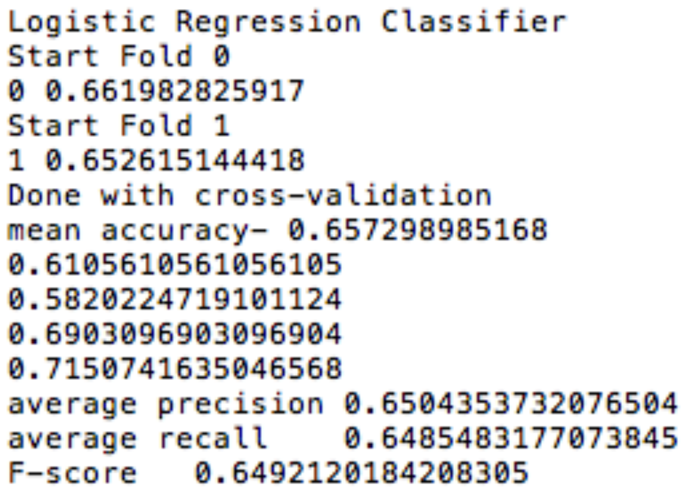
```
LIWCsets_with_preprocessing-
Naive Bayes Classifier
Start Fold 0
0 0.6428571428571429
Start Fold 1
1 0.6342701014832163
Done with cross-validation
mean accuracy- 0.6385636221701796
0.6009745533297238
0.49887640449438203
0.6597497711321331
0.7457744049672301
average precision 0.6303621622309284
average recall    0.6223254047308061
F-score   0.6226580870751346


Logistic Regression Classifier
Start Fold 0
0 0.661982825917
Start Fold 1
1 0.65612802498
Done with cross-validation
mean accuracy- 0.659055425449
0.6129489603024575
0.5829213483146067
0.6914893617021277
0.7174887892376681
average precision 0.6522191610022926
average recall    0.6502050687761374
F-score   0.6509036810952159
```

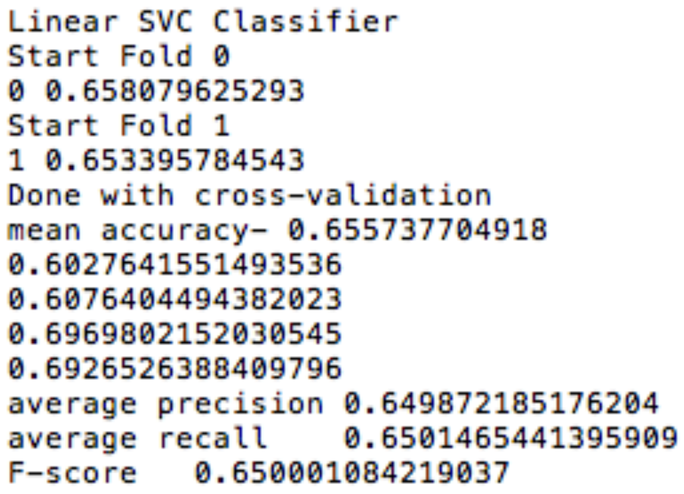
SL + LIWCsets with Preprocessing:-



```
SL + LIWCsets_with_preprocessing
Naive Bayes Classifier
Start Fold 0
0 0.6440281030444965
Start Fold 1
1 0.6323185011709602
Done with cross-validation
mean accuracy- 0.6381733021077284
0.6005420054200542
0.49797752808988766
0.65934736200061
0.7457744049672301
average precision 0.6299446837103321
average recall    0.6218759665285589
F-score   0.6221873128315343
```

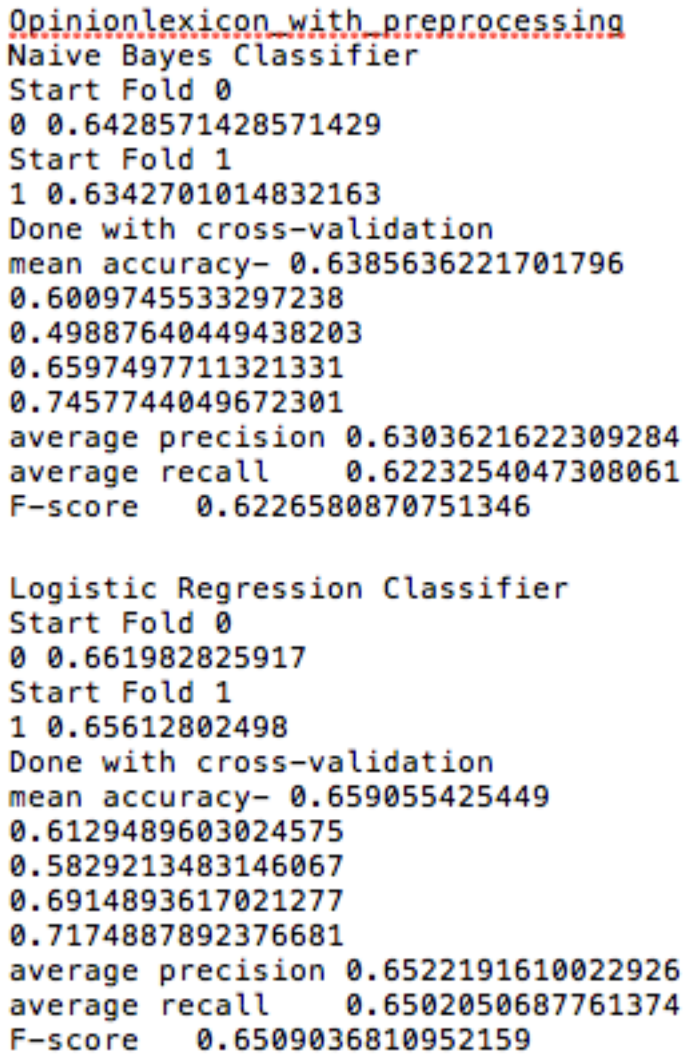


```
Logistic Regression Classifier
Start Fold 0
0 0.661982825917
Start Fold 1
1 0.652615144418
Done with cross-validation
mean accuracy- 0.657298985168
0.6105610561056105
0.5820224719101124
0.6903096903096904
0.7150741635046568
average precision 0.6504353732076504
average recall    0.6485483177073845
F-score   0.6492120184208305
```



```
Linear SVC Classifier
Start Fold 0
0 0.658079625293
Start Fold 1
1 0.653395784543
Done with cross-validation
mean accuracy- 0.655737704918
0.6027641551493536
0.6076404494382023
0.6969802152030545
0.6926526388409796
average precision 0.649872185176204
average recall    0.6501465441395909
F-score   0.650001084219037
```

Opinionlexicon with Preprocessing:-



```
Opinionlexicon_with_preprocessing
Naive Bayes Classifier
Start Fold 0
0 0.6428571428571429
Start Fold 1
1 0.6342701014832163
Done with cross-validation
mean accuracy- 0.6385636221701796
0.6009745533297238
0.49887640449438203
0.6597497711321331
0.7457744049672301
average precision 0.6303621622309284
average recall    0.6223254047308061
F-score   0.6226580870751346

Logistic Regression Classifier
Start Fold 0
0 0.661982825917
Start Fold 1
1 0.65612802498
Done with cross-validation
mean accuracy- 0.659055425449
0.6129489603024575
0.5829213483146067
0.6914893617021277
0.7174887892376681
average precision 0.6522191610022926
average recall    0.6502050687761374
F-score   0.6509036810952159
```

Accuracy:-

Sentences:- 5000+
No of folds = 2

a.) Without pre-processing

	Naïve Bayes	Multi Nomial NB	Bernoulli NB	Decision Tree	Logistic Regression	SGDC	Linear SVC	Random Forest	**Avera (Featu**
Unigram	0.6393	0.6191	0.648	0.608	0.6729	0.6360	0.6144	0.6147	**0.63905**
Bigram	0.6479	0.6178	0.6461	0.6135	0.6131	0.5936	0.6160	0.6055	**0.63418**
POS	0.6453	0.6139	0.6461	0.6061	0.6344	0.5169	0.6111	0.6188	**0.61907**
Negation	0.5311	0.660	0.5686	0.604	0.6425	0.5792	0.6210	0.5497	**0.60326**
SL	0.6506	0.6174	0.6494	0.6026	0.6750	0.5977	0.671	0.6329	**0.63707**
LIWC	0.6467	0.6247	0.6457	0.5987	0.6048	0.6252	0.6050	0.6202	**0.63887**
SL+ LIWC	0.6502	0.622	0.6491	0.6055	0.6234	0.5228	0.6752	0.6088	**0.62587**
Opinion	0.6467	0.6247	0.6457	0.6028	0.6148	0.6360	0.6558	0.615	**0.641**
Average (Classifier-wise)	**0.632225**	**0.62495**	**0.6373375**	**0.60515**	**0.6738625**	**0.588425**	**0.6674375**	**0.6090125**	

b). With pre-processing

	Naïve Bayes	Multi Nomial NB	Bernoulli NB	Decision Tree	Logistic Regression	SGDC	Linear SVC	Random Forest	**Avera (Featu**
Unigram	0.6368	0.6381	0.63622	0.58196	0.6994	0.6209	0.6551	0.6044	**0.6491**
Bigram	0.6389	0.6354	0.6358	0.5815	0.6796	0.6247	0.6659	0.5878	**0.62745**
POS	0.6366	0.6317	0.6282	0.5839	0.6802	0.4426	0.6983	0.5936	**0.5968**
Negation	0.6225	0.5895	0.5823	0.6010	0.6607	0.5764	0.6587	0.5745	**0.60445**
SL	0.6348	0.6383	0.6362	0.5737	0.66674	0.5899	0.6599	0.6018	**0.624**
LIWC	0.6385	0.6375	0.6356	0.5848	0.6390	0.6053	0.6878	0.5977	**0.6270**
SL+ LIWC	0.6381	0.6360	0.6379	0.5778	0.7072	0.5749	0.6857	0.6024	**0.6425**
Opinion	0.6385	0.6375	0.6356	0.5891	0.6990	0.6059	0.6876	0.6135	**0.6395**
Average (Classifier-wise)	**0.6355**	**0.6305**	**0.6284775**	**0.58422**	**0.6985625**	**0.5800**	**0.6886**	**0.59696**	

Observation for the Tables:-

I observe both the tables as Pre-process data and without pre-processed data. At the end the Pre-processed data gave more accuracy than without pre-processed data because we already removed the stopwords and other unnecessary words from the text.

- **The Logistic Regression and Linear SVC gave the best performance in terms of accuracy in compare to other algorithms.**

- Naïve Bayes, MultiNomial NB and Bernoulli NB gave some what similar perfornce on average.

- The other three as Decision Tree, SGDC and Random Forest gave the lower performace compare to others.

- On Average we got the score of F-score and Precision as 0.63 which I think as a very good margin to get.

- On feature Set I got the best performance on average as Unigram and SL + LIWC on every algorithm.

- As well as Opinion set also gave a better performance in compare to others.

Conclusion:-

- At the end I would like to say as in individual accuracy I got the mark up **to 0.75 on Logistic Regression** and on average 0.698 **~ 0.70 as the limit of 5000+** sentences. I have also calculated the review text accuracy of year 2009 and 2010. In that I got around 0.59 for 2009 for 1661 sentences and 0.65 for 3500 sentences for 2010. My prediction as If I have gone through 15000+ sentences then we could have got more than 0.85 accuracy.
- So My conclusion is the more data we have in out featureset. We could get better results from comparing more words to our data set.

- The Screenshots I have already add above topic wise.

Instructions to execute:-

- First the File HW4.ipynb file to get the tags for sentences as 'pos' and 'neg'. (Lastly I used upto 5000 common words to compare with)
- Then one folder as AmazonReviews would be there. In that folder classifyAmazon.py file to execute. There are other py files which are connected to that one file so keep it in that folder only.
- The execution for this may take more than 3 to 4 hours. Just for your Information.
- The output files are there in the folder as Output Files (Final_Output).
- The CSV Files are there for 2009,2010 and combined in one folder as CSV Files.
- I am also attaching the screenshots in another folder just to check.
- All the data the grader can find in Corpus folder.