Processing and classification of sentiment on Twitter dataset

Ayesha Ahmad

251665317

MS in Computer science

Abstract

Twitter is an open online platform for interaction with the world. It is social network where users have access to news and media through a type of message named as tweets. Tweets were originally messages of max length of 140 character. Users were to express their thoughts and messages in 140 characters. Later, media functionality was added to it, which allowed users to post pictures, polls, videos, etc. In November 2017, the maximum length was increased to 280 characters. For the purpose of this analysis we will be using Twitter dataset from 2014. Twitter is a very original idea that was launched in 2006. It was not very clear what twitter was, when the originators brainstormed the idea of twitter. Microblogging, social network, news, many uses but one name, Twitter! As of October 2017, there are 330 million active users on Twitter, constantly creating and sharing data. This data is being mined for several uses. Twitter data is a very good resource to understand what the society is experiencing. In fact, on the day of the 2016 U.S. presidential election, Twitter proved to be the largest source of breaking news, with 40 million election-related tweets sent by 10 p.m. (Eastern Time) that day. Sentiment Analysis is the process of 'computationally' determining whether a piece of writing is positive, negative or neutral. It's also known as opinion mining, deriving the opinion or attitude of a speaker. For this project, a sentiment classifier is created with the help of different techniques (different feature set, different techniques of preprocessing, libraries) and the results are compared.

Data Processing

Data

For the purpose of this project, we use Twitter dataset created by SemEval.

SemEval shared task Twitter data, labeled for sentiment. The Semantic Evaluation conference, SemEval, has recently added sentiment detection in Twitter messages and other social media genres to its shared tasks.

For example, in 2014, it ran as Task 9, sub-task B, Message Polarity Classification: http://alt.qcri.org/semeval2014/task9/.

The data was manually labeled and each dataset is available as a Twitter id paired with the manual label. There are 5 labels used for each message: **"positive", "negative", "objective", "neutral", "objective OR neutral"**. For this series of experiments, three class, "pos", "neg" and "neu" are chosen. To actually get the tweets, a script can be run to get Tweets from Twitter. If a Twitter user has retracted their tweet, then Twitter will no longer send it out and it is marked as "Not Available". The dataset given here was collected in the Spring 2014 from Twitter.

Processing steps

The tweets dataset need to be preprocessed, to remove noise. For example, stop words, differently cased words, etc. Following is the order of steps used to preprocess the tweets, for one of the experiments:

1. Convert every word to lower.
2. Apply filter for alphabets determined by the regular expression: ^[^a-z] +$
3. Apply filter for stop words- The list of stop words is obtained from 'stopwords_twitter.txt'

For the other experiments, words are converted to lower case.

Features

Experiments are performed with three kinds of feature sets. Document, SL, and POS
To create the feature set, all the words are taken into consideration. In this case, it is recommended to not filter any words for these. However, as one of the experiments, it is chosen to create feature set using the filtered document.

Document/Bag of words feature

In a bag of words feature, all the words in the corpus are collected and some number of most frequent words are selected to be the word features.

Subjectivity Lexicon

This feature set defines features that include word counts of subjectivity words negative feature will have number of weakly negative words + 2 * number of strongly negative words. positive feature has similar definition, not counting neutral words.

Part of Speech

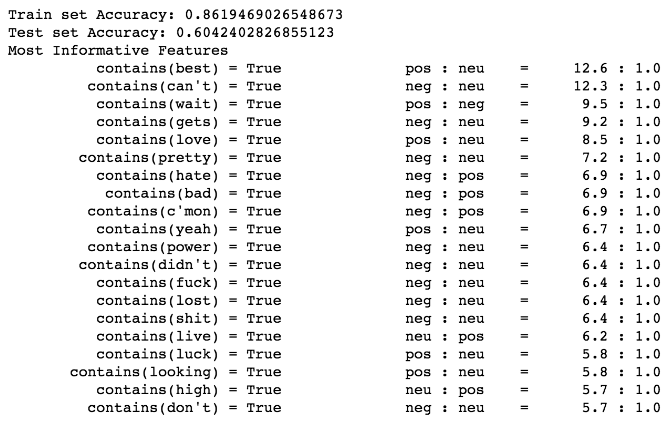
The part of speech tagger feature set takes a document list of words and returns a feature dictionary which it gets by running the default POS tagger (the Stanford tagger) on the document and counting four types of POS tags to use as features

Experiments and Results

Filtered Document vs. Non-filtered document feature set

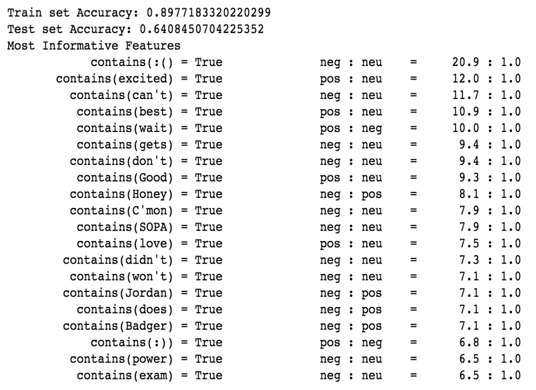
In the first experiment, the feature set is obtained from filtered twitter document and non-filtered twitter document by applying document type feature set on them. Explanation for document feature set can be found in the features section.

Figure 1 shows result of document features on preprocessed dataset. Figure 2 shows the result of document features on non-filtered dataset. The experiment shows a train accuracy of 0.91 for document feature on preprocessed tweets vs 0.89 for document feature on all tweets. However, it shows a test accuracy of 0.61 on filtered tweets and 0.64 for non-filtered tweets. Thus, for the experiments going ahead, all tweets are considered for creating feature sets. Also, shown in the figure are 20 most informative features. Both have similar set of mostly occurring features. However, some uncommon ones are luck, power, etc.



```
Train set Accuracy: 0.8619469026548673
Test set Accuracy: 0.6042402826855123
Most Informative Features
           contains(best) = True              pos : neu    =     12.6 : 1.0
          contains(can't) = True              neg : neu    =     12.3 : 1.0
           contains(wait) = True              pos : neg    =      9.5 : 1.0
           contains(gets) = True              neg : neu    =      9.2 : 1.0
           contains(love) = True              pos : neu    =      8.5 : 1.0
         contains(pretty) = True              neg : neu    =      7.2 : 1.0
           contains(hate) = True              neg : pos    =      6.9 : 1.0
            contains(bad) = True              neg : pos    =      6.9 : 1.0
          contains(c'mon) = True              neg : pos    =      6.9 : 1.0
           contains(yeah) = True              pos : neu    =      6.7 : 1.0
          contains(power) = True              neg : neu    =      6.4 : 1.0
         contains(didn't) = True              neg : neu    =      6.4 : 1.0
           contains(fuck) = True              neg : neu    =      6.4 : 1.0
           contains(lost) = True              neg : neu    =      6.4 : 1.0
           contains(shit) = True              neg : neu    =      6.4 : 1.0
           contains(live) = True              neu : pos    =      6.2 : 1.0
           contains(luck) = True              pos : neu    =      5.8 : 1.0
        contains(looking) = True              pos : neu    =      5.8 : 1.0
           contains(high) = True              neu : pos    =      5.7 : 1.0
          contains(don't) = True              neg : neu    =      5.7 : 1.0
```

Figure 1:Document feature on preprocessed (and filtered) tweets

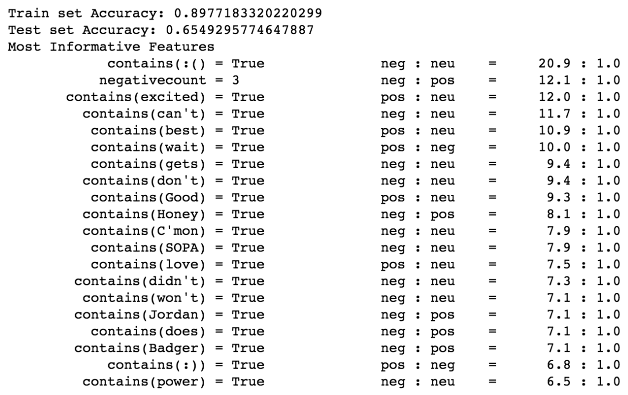


```
Train set Accuracy: 0.8977183320220299
Test set Accuracy: 0.6408450704225352
Most Informative Features
             contains(:() = True              neg : neu    =     20.9 : 1.0
        contains(excited) = True              pos : neu    =     12.0 : 1.0
          contains(can't) = True              neg : neu    =     11.7 : 1.0
           contains(best) = True              pos : neu    =     10.9 : 1.0
           contains(wait) = True              pos : neg    =     10.0 : 1.0
           contains(gets) = True              neg : neu    =      9.4 : 1.0
          contains(don't) = True              neg : neu    =      9.4 : 1.0
           contains(Good) = True              pos : neu    =      9.3 : 1.0
          contains(Honey) = True              neg : pos    =      8.1 : 1.0
          contains(C'mon) = True              neg : neu    =      7.9 : 1.0
           contains(SOPA) = True              neg : neu    =      7.9 : 1.0
           contains(love) = True              pos : neu    =      7.5 : 1.0
         contains(didn't) = True              neg : neu    =      7.3 : 1.0
          contains(won't) = True              neg : neu    =      7.1 : 1.0
         contains(Jordan) = True              neg : pos    =      7.1 : 1.0
           contains(does) = True              neg : pos    =      7.1 : 1.0
         contains(Badger) = True              neg : pos    =      7.1 : 1.0
             contains(:)) = True              pos : neg    =      6.8 : 1.0
          contains(power) = True              neg : neu    =      6.5 : 1.0
           contains(exam) = True              neg : neu    =      6.5 : 1.0
```

Figure 2:Document feature on all tweets

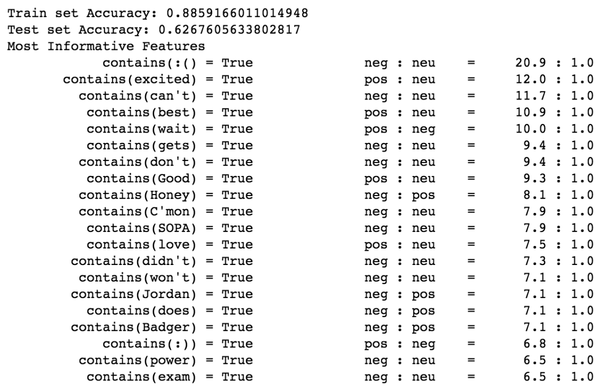
Document vs Subjectivity Lexicon vs Part of speech feature set

Document feature set (Figure 2), SL feature set (Figure 3) and POS feature set(Figure 4) are created from all tweets. A test accuracy of 0.64, 0.65 and 0.62 are obtained for the feature sets respectively. A training accuracy of 0.89, 0.89, 0.88 are obtained, respectively. They all have similar performance, however the testing accuracy of SL



```
Train set Accuracy: 0.8977183320220299
Test set Accuracy: 0.6549295774647887
Most Informative Features
             contains(:() = True              neg : neu    =     20.9 : 1.0
            negativecount = 3                 neg : pos    =     12.1 : 1.0
        contains(excited) = True              pos : neu    =     12.0 : 1.0
          contains(can't) = True              neg : neu    =     11.7 : 1.0
           contains(best) = True              pos : neu    =     10.9 : 1.0
           contains(wait) = True              pos : neg    =     10.0 : 1.0
           contains(gets) = True              neg : neu    =      9.4 : 1.0
          contains(don't) = True              neg : neu    =      9.4 : 1.0
           contains(Good) = True              pos : neu    =      9.3 : 1.0
          contains(Honey) = True              neg : pos    =      8.1 : 1.0
          contains(C'mon) = True              neg : neu    =      7.9 : 1.0
           contains(SOPA) = True              neg : neu    =      7.9 : 1.0
           contains(love) = True              pos : neu    =      7.5 : 1.0
         contains(didn't) = True              neg : neu    =      7.3 : 1.0
          contains(won't) = True              neg : neu    =      7.1 : 1.0
         contains(Jordan) = True              neg : pos    =      7.1 : 1.0
           contains(does) = True              neg : pos    =      7.1 : 1.0
         contains(Badger) = True              neg : pos    =      7.1 : 1.0
             contains(:)) = True              pos : neg    =      6.8 : 1.0
          contains(power) = True              neg : neu    =      6.5 : 1.0
```

Figure 3: SL Feature set



```
Train set Accuracy: 0.8859166011014948
Test set Accuracy: 0.6267605633802817
Most Informative Features
             contains(:() = True              neg : neu    =     20.9 : 1.0
        contains(excited) = True              pos : neu    =     12.0 : 1.0
          contains(can't) = True              neg : neu    =     11.7 : 1.0
           contains(best) = True              pos : neu    =     10.9 : 1.0
           contains(wait) = True              pos : neg    =     10.0 : 1.0
           contains(gets) = True              neg : neu    =      9.4 : 1.0
          contains(don't) = True              neg : neu    =      9.4 : 1.0
           contains(Good) = True              pos : neu    =      9.3 : 1.0
          contains(Honey) = True              neg : pos    =      8.1 : 1.0
          contains(C'mon) = True              neg : neu    =      7.9 : 1.0
           contains(SOPA) = True              neg : neu    =      7.9 : 1.0
           contains(love) = True              pos : neu    =      7.5 : 1.0
         contains(didn't) = True              neg : neu    =      7.3 : 1.0
          contains(won't) = True              neg : neu    =      7.1 : 1.0
         contains(Jordan) = True              neg : pos    =      7.1 : 1.0
           contains(does) = True              neg : pos    =      7.1 : 1.0
         contains(Badger) = True              neg : pos    =      7.1 : 1.0
             contains(:)) = True              pos : neg    =      6.8 : 1.0
          contains(power) = True              neg : neu    =      6.5 : 1.0
           contains(exam) = True              neg : neu    =      6.5 : 1.0
```

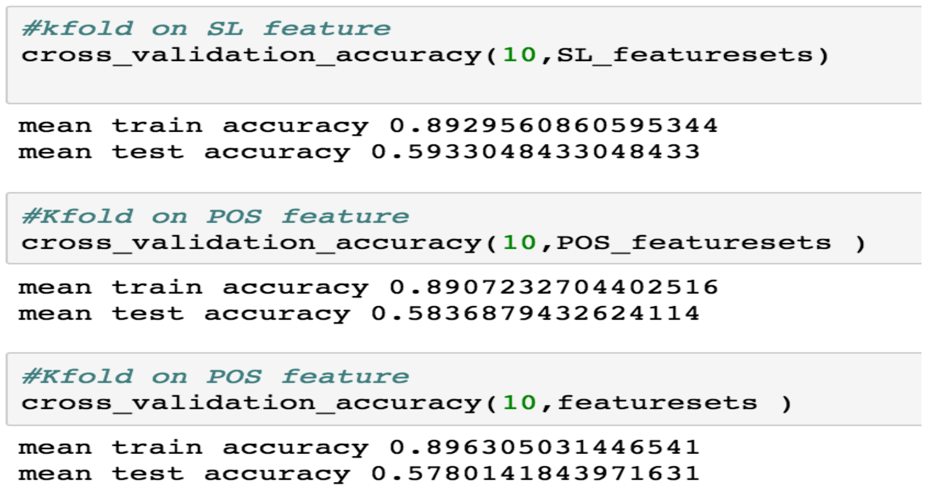
Figure 4:POS feature set

feature set is highest. So, for the remaining experiments, SL feature sets are chosen. Also, it can be seen from the experiments done so far, that accuracies of training sets are much higher than accuracy of test set. This shows that there is a level of overfitting that is happening in the classifier models. Thus, there is an attempt to improve the model performance on test set by using k folds approach discussed and experimented with in the next section.

Naïve Bayes classifier without cross validation vs Naïve Bayes classifier without K-fold cross validation

Cross-validation, sometimes called rotation estimation, is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set. It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. In a prediction problem, a model is usually given a dataset of known data on which training is run (training dataset), and a dataset of unknown data (or first seen data) against which the model is tested (called the validation dataset or testing set). The goal of cross validation is to define a dataset to "test" the model in the training phase (i.e., the validation set), in order to limit problems like overfitting, give an insight on how the model will generalize to an independent dataset.

In k-fold cross-validation, the original sample is randomly partitioned into k equal sized subsamples. Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining k − 1 subsamples are used as training data. The cross-validation process is then repeated k times (the folds), with each of the k subsamples used exactly once as the validation data. The k results from the folds can then be averaged to produce a single estimation. The advantage of this method over repeated random sub-sampling is that all observations are used for both training and validation, and each observation is used for validation exactly once. 10-fold cross-validation is commonly used, but in general k remains an unfixed parameter.



```
#kfold on SL feature
cross_validation_accuracy(10,SL_featuresets)
```

```
mean train accuracy 0.8929560860595344
mean test accuracy 0.5933048433048433
```

```
#Kfold on POS feature
cross_validation_accuracy(10,POS_featuresets )
```

```
mean train accuracy 0.8907232704402516
mean test accuracy 0.5836879432624114
```

```
#Kfold on POS feature
cross_validation_accuracy(10,featuresets )
```

```
mean train accuracy 0.896305031446541
mean test accuracy 0.5780141843971631
```

Figure 5:k fold on sl, pos, document(typo in the last execution comment)

In this experiment, the aim is to get a better test accuracy by using cross validation. Thus, cross validation is performed on all three feature sets, document, subjective lexicon, part of speech feature sets. Though the accuracy itself is not as good as one would hope for, the use of new metrics known as precision, recall and f score.

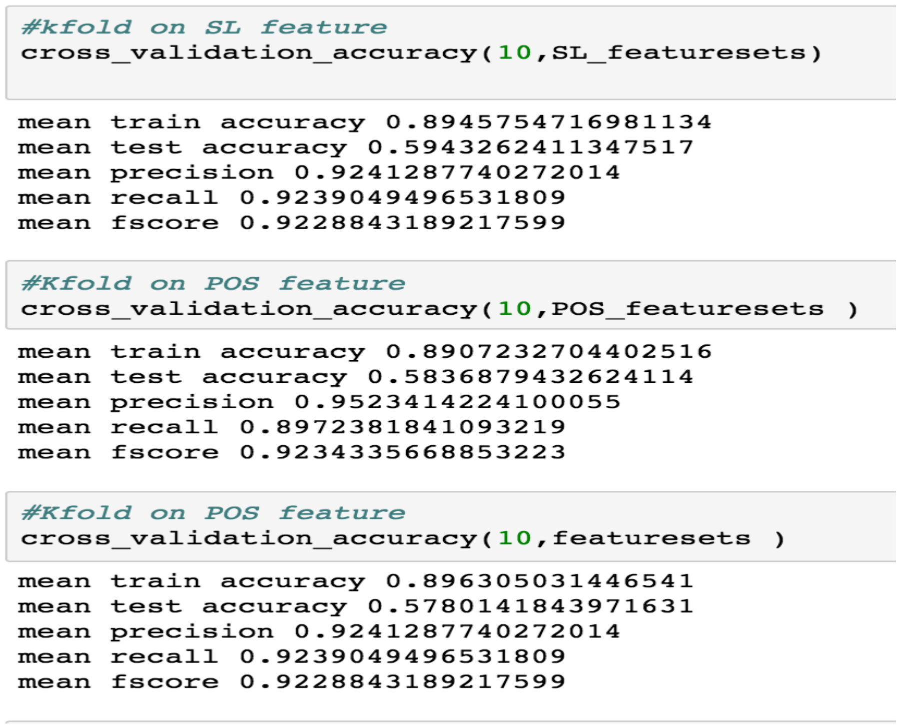
Precision (also called positive predictive value) is the fraction of relevant instances among the retrieved instances

Recall (also known as sensitivity) is the fraction of relevant instances that have been retrieved over the total number of relevant instances.

Both precision and recall are therefore based on an understanding and measure of relevance.

A measure that combines precision and recall is the harmonic mean of precision and recall, the traditional F-measure or balanced F-score:

$$F = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$



```
#kfold on SL feature
cross_validation_accuracy(10,SL_featuresets)

mean train accuracy 0.8945754716981134
mean test accuracy 0.5943262411347517
mean precision 0.9241287740272014
mean recall 0.9239049496531809
mean fscore 0.9228843189217599

#Kfold on POS feature
cross_validation_accuracy(10,POS_featuresets )

mean train accuracy 0.8907232704402516
mean test accuracy 0.5836879432624114
mean precision 0.9523414224100055
mean recall 0.8972381841093219
mean fscore 0.9234335668853223

#Kfold on POS feature
cross_validation_accuracy(10,featuresets )

mean train accuracy 0.896305031446541
mean test accuracy 0.5780141843971631
mean precision 0.9241287740272014
mean recall 0.9239049496531809
mean fscore 0.9228843189217599
```

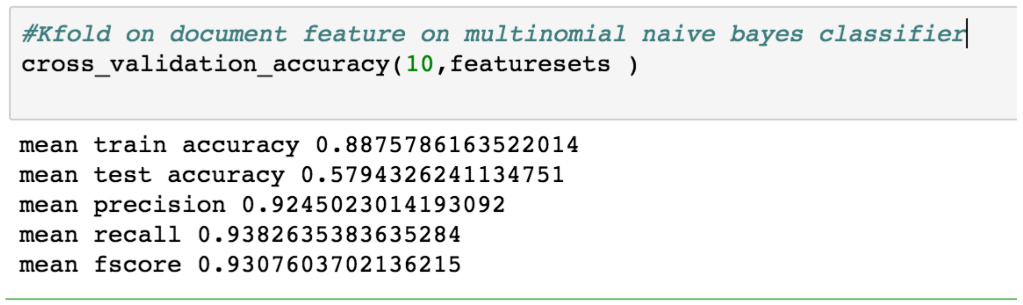
Figure 6:k fold on sl, pos, document(typo in the last execution comment)

As it can be seen from the above screen shot and description, that precision, recall and f-measure give phenomenal scores as opposed to accuracy. They are more commonly used to certify the robustness of models.

NLTK Naïve Bayes Classifier vs SK-learn Multinomial Naïve Bayes Classifier

For this last experiment Scikit-learn's Multinomial Naïve Bayes algorithm has been chosen to compare with NLTK's Naïve Bayes algorithm

The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.



```
#Kfold on document feature on multinomial naive bayes classifier
cross_validation_accuracy(10,featuresets )
```

```
mean train accuracy 0.8875786163522014
mean test accuracy 0.5794326241134751
mean precision 0.9245023014193092
mean recall 0.9382635383635284
mean fscore 0.9307603702136215
```

By referring to this and output for nltk naïve Bayes classifier, it is seen that there are negligible differences between the performance of the classifiers from nltk and sklearn.

Conclusion

When the results of all these classification models are compared, it can be seen that there is not a major difference in the performance measures of them all. For future work, some additional features may be picked up. Entity extraction can improve the feature set and performance result leading to a better model. This was out of scope for this project, however, it would be a good extension to take up.

Bibliography

1. Martinez-Camara, E., Martin-Valdivia, M., Urena-Lopez, L., & Montejo-Raez, A. (2014;2012;). Sentiment analysis in twitter. Natural Language Engineering, 20(1), 1-28. doi:10.1017/S1351324912000332
2. Kiritchenko, S., Zhu, X., & Mohammad, S. (2014). Sentiment analysis of short informal text. Journal of Artificial Intelligence Research, 50, 723-762.
3. Saif, H., He, Y., Fernandez, M., & Alani, H. (2016). Contextual semantics for sentiment analysis of twitter. Information Processing & Management, 52(1), 5-19. doi:10.1016/j.ipm.2015.01.005

Appendix: Code script

```
'''
  This program shell reads tweet data for the twitter sentiment classification
problem.
  The input to the program is the path to the Semeval directory "corpus" and a limit
number.
  The program reads the first limit number of tweets
  It creates a "tweetdocs" variable with a list of tweets consisting of a pair
    with the list of tokenized words from the tweet and the label pos, neg or neu.
  It prints a few example tweets, as text and as tokens.
  Your task is to generate features sets and train and test a classifier.

  Usage:  python classifySemevalTweets.py  <corpus directory path> <limit number>
'''
# open python and nltk packages needed for processing
# while the semeval tweet task b data has tags for "positive", "negative",
#  "objective", "neutral", "objective-OR-neutral", we will combine the last 3 into
"neutral"
import nltk
import sklearn
import csv
import numpy as np
import pandas as pd
import textblob
import numpy
from nltk.corpus import sentence_polarity
import os
import sys
import nltk
from nltk.tokenize import TweetTokenizer
from nltk.metrics import ConfusionMatrix
twtokenizer = TweetTokenizer()
import re
import math

# create a path to where the subjectivity file resides on your disk

# create your own path to the subjclues file
SLpath = "subjclueslen1-HLTEMNLP05.tff"


# read stop words from file if used
stopwords = open('stopwords_twitter.txt', 'r')
stoptokens = nltk.word_tokenize(stopwords.read())

# function that takes a word and returns true if it consists only
#   of non-alphabetic characters  (assumes import re)
def alpha_filter(w):
  # pattern to match word of non-alphabetical characters
    pattern = re.compile('^[^a-z]+$')
    if (pattern.match(w)):
        return True
    else:
        return False
# define a feature definition function here
# feature sets from a feature definition function

# define features (keywords) of a document for a BOW/unigram baseline
# each feature is 'contains(keyword)' and is true or false depending
# on whether that keyword is in the document
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features
####   adding features from a sentiment lexicon   ####
# First look at the program in the file Subjectivity.py to load the subjectivity
lexicon
# copy and paste the definition of the readSubjectivity function

# this function returns a dictionary where you can look up words and get back
#     the four items of subjectivity information described above
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
# define features that include word counts of subjectivity words
# negative feature will have number of weakly negative words +
#    2 * number of strongly negative words
# positive feature has similar definition
#    not counting neutral words
SL = readSubjectivity(SLpath)
# how many words are in the dictionary
len(SL.keys())
def SL_features(document, word_features, SL):
    document_words = set(document)
    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)
    return features
```

```
# this function takes a document list of words and returns a feature dictionary
# it runs the default pos tagger (the Stanford tagger) on the document
#   and counts 4 types of pos tags to use as features
def POS_features(document, word_features):
    document_words = set(document)
    tagged_words = nltk.pos_tag(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    numNoun = 0
    numVerb = 0
    numAdj = 0
    numAdverb = 0
    for (word, tag) in tagged_words:
        if tag.startswith('N'): numNoun += 1
        if tag.startswith('V'): numVerb += 1
        if tag.startswith('J'): numAdj += 1
        if tag.startswith('R'): numAdverb += 1
    features['nouns'] = numNoun
    features['verbs'] = numVerb
    features['adjectives'] = numAdj
    features['adverbs'] = numAdverb
    return features
# function for CV scores
def CV_Scores(classifier_type, test_set):
    from collections import Counter
    from nltk.metrics import ConfusionMatrix
    refList = []
    testList = []
    for features, label in test_set:
        refList.append(label)
        testList.append(classifier_type.classify(features))
    cm = ConfusionMatrix(refList, testList)
    true_positives = Counter()
    false_negatives = Counter()
    false_positives = Counter()
    for i in refList:
        for j in refList:
            if i == j:
                true_positives[i] += cm[i,j]
            else:
                false_negatives[i] += cm[i,j]
                false_positives[j] += cm[i,j]
    #print "Confusion Matrix:"
    #print cm
    precision=0
    recall=0
    fscore=0
    for i in sorted(refList):
        if true_positives[i] == 0:
            fscore = 0
        else:
            precision = true_positives[i] / 
float(true_positives[i]+false_positives[i])
            recall = true_positives[i] / float(true_positives[i]+false_negatives[i])
            fscore = 2 * (precision * recall) / float(precision + recall)
    return precision, recall , fscore
#cross validation
## cross-validation ##
# this function takes the number of folds, the feature sets
# it iterates over the folds, using different sections for training and testing in 
turn
#   it prints the accuracy for each fold and the average accuracy at the end
```

```
def cross_validation_accuracy(num_folds, featuresets):
    subset_size = int(len(featuresets)/num_folds)
    accuracy_train_list = []
    accuracy_test_list = []
    presion_list=[]
    recall_list=[]
    fscore_list=[]
    # iterate over the folds
    for i in range(num_folds):
        test_this_round = featuresets[(i*subset_size):][:subset_size]
        train_this_round = featuresets[:(i*subset_size)] +
featuresets[((i+1)*subset_size):]
        # train using train_this_round
        classifier = nltk.NaiveBayesClassifier.train(train_this_round)
        # evaluate against test_this_round and save accuracy
        accuracy_train_this_round = nltk.classify.accuracy(classifier,
train_this_round)
        accuracy_test_this_round = nltk.classify.accuracy(classifier, test_this_round)
        accuracy_train_list.append(accuracy_train_this_round)
        accuracy_test_list.append(accuracy_test_this_round)
        precision, recall,fscore =CV_Scores(classifier, test_this_round)
        presion_list.append(precision)
        recall_list.append(recall)
        fscore_list.append(fscore)
    # find mean accuracy over all rounds
    print ('mean train accuracy', sum(accuracy_train_list) / num_folds)
    print ('mean test accuracy', sum(accuracy_test_list) / num_folds)
    print ('mean precision', sum(presion_list) / num_folds)
    print ('mean recall', sum(recall_list) / num_folds)
    print ('mean fscore', sum(fscore_list) / num_folds)
## cross-validation ##
# this function takes the number of folds, the feature sets
# it iterates over the folds, using different sections for training and testing in
turn
#   it prints the accuracy for each fold and the average accuracy at the end
def cross_validation_accuracy_SKlearn(num_folds, featuresets):
    subset_size = int(len(featuresets)/num_folds)
    accuracy_train_list = []
    accuracy_test_list = []
    presion_list=[]
    recall_list=[]
    fscore_list=[]
    # iterate over the folds
    for i in range(num_folds):
        test_this_round = featuresets[(i*subset_size):][:subset_size]
        train_this_round = featuresets[:(i*subset_size)] +
featuresets[((i+1)*subset_size):]
        # train using train_this_round
        classifier = SklearnClassifier(MultinomialNB())
        classifier.train(train_this_round)

        # evaluate against test_this_round and save accuracy
        accuracy_train_this_round = nltk.classify.accuracy(classifier,
train_this_round)
        accuracy_test_this_round = nltk.classify.accuracy(classifier, test_this_round)
        accuracy_train_list.append(accuracy_train_this_round)
        accuracy_test_list.append(accuracy_test_this_round)
        precision, recall,fscore =CV_Scores(classifier, test_this_round)
        presion_list.append(precision)
        recall_list.append(recall)
        fscore_list.append(fscore)
    # find mean accuracy over all rounds
    print ('mean train accuracy', sum(accuracy_train_list) / num_folds)
    print ('mean test accuracy', sum(accuracy_test_list) / num_folds)
    print ('mean precision', sum(presion_list) / num_folds)
    print ('mean recall', sum(recall_list) / num_folds)
    print ('mean fscore', sum(fscore_list) / num_folds)
# function to read tweet training file, train and test a classifier
def processtweets(dirPath,limitStr):
  # convert the limit argument from a string to an int
  limit = int(limitStr)
  # initialize NLTK built-in tweet tokenizer
  twtokenizer = TweetTokenizer()

  os.chdir(dirPath)

  f = open('./downloaded-tweeti-b-dist.tsv', 'r')
  # loop over lines in the file and use the first limit of them
  #    assuming that the tweets are sufficiently randomized
  tweetdata = []
  for line in f:
    if (len(tweetdata) < limit):
      # remove final end of line character
      line = line.strip()
      # each line has 4 items separated by tabs
      # ignore the tweet and user ids, and keep the sentiment and tweet text
      tweetdata.append(line.split('\t')[2:4])

  for tweet in tweetdata[:10]:
    print (tweet)

  # create list of tweet documents as (list of words, label)
  # where the labels are condensed to just 3:  'pos', 'neg', 'neu'
  tweetdocs = []
  # add all the tweets except the ones whose text is Not Available
  for tweet in tweetdata:
    if (tweet[1] != 'Not Available'):
      # run the tweet tokenizer on the text string - returns unicode tokens, so
convert to utf8
      tokens = twtokenizer.tokenize(tweet[1])

      if tweet[0] == '"positive"':
        label = 'pos'
      else:
        if tweet[0] == '"negative"':
          label = 'neg'
        else:
          if (tweet[0] == '"neutral"') or (tweet[0] == '"objective"') or (tweet[0] ==
'"objective-OR-neutral"'):
            label = 'neu'
          else:
            label = ''
      tweetdocs.append((tokens, label))

  # print a few
  for tweet in tweetdocs[:10]:
    print (tweet)

  # possibly filter tokens
  # first preprocessing and then possibly filter tokens
    processed_tweets=[]
    for tweet,label in tweetdocs:
        x= [w.lower() for w in tweet]
        x= [w for w in x if not alpha_filter(w)]
        x= [w for w in x if not w in stoptokens]
        processed_tweets.append((x,label))
  # continue as usual to get all words and create word features
  # continue as usual to get all words and create word features

# get all words from all tweet and put into a frequency distribution
#   note lowercase, but no stemming or stopwords
    all_words_list = [word for (tweet,label) in tweetdocs for word in tweet]

    print(all_words_list[:100])
    all_words = nltk.FreqDist(all_words_list)
    # get the 2000 most frequently appearing keywords in the corpus
    word_items = all_words.most_common(2000)
    word_features = [word for (word,count) in word_items]
  # feature sets from a feature definition function
  # get features sets for a document, including keyword features and category feature
    featuresets_processed = [(document_features(d, word_features), c) for (d, c) in
processed_tweets]
  # get features sets for a document, including keyword features and category feature
    featuresets = [(document_features(d, word_features), c) for (d, c) in tweetdocs]
    SL_featuresets = [(SL_features(d, word_features, SL), c) for (d, c) in tweetdocs]
    POS_featuresets = [(POS_features(d, word_features), c) for (d, c) in tweetdocs]
    # train and test a classifier
    print("********Document Feature with filtered tweets*******")
    num_docs = len(featuresets_processed)
    #to create 80-10 split
    train_set, test_set =
featuresets_processed[:math.floor(0.8*num_docs)],featuresets_processed[math.floor(0.8*
num_docs):]
    classifier = nltk.NaiveBayesClassifier.train(train_set)
    print("Train set Accuracy: "+ str(nltk.classify.accuracy(classifier, train_set)))
    print("Test set Accuracy: "+ str(nltk.classify.accuracy(classifier, test_set)))
    # show most informative features
    print(classifier.show_most_informative_features(20))

    print("********Document Feature with all tweets*******")
    # train and test a classifier
    num_docs = len(featuresets)
    #to create 90-10 split
    train_set, test_set =
featuresets[:math.floor(0.9*num_docs)],featuresets[math.floor(0.9*num_docs):]
    classifier = nltk.NaiveBayesClassifier.train(train_set)
    print("Train set Accuracy: "+ str(nltk.classify.accuracy(classifier, train_set)))
    print("Test set Accuracy: "+ str(nltk.classify.accuracy(classifier, test_set)))
    # show most informative features
    print(classifier.show_most_informative_features(20))

    print("********SL Feature with all tweets*******")

    num_docs = len(featuresets)
    #to create 90-10 split
    train_set, test_set =
SL_featuresets[:math.floor(0.9*num_docs)],SL_featuresets[math.floor(0.9*num_docs):]
    # retrain the classifier using these features
    classifier = nltk.NaiveBayesClassifier.train(train_set)
    print("Train set Accuracy: "+ str(nltk.classify.accuracy(classifier, train_set)))
    print("Test set Accuracy: "+ str(nltk.classify.accuracy(classifier, test_set)))
    # show most informative features
    print(classifier.show_most_informative_features(20))
    print("********POS Feature with all tweets*******")

    # train and test the classifier
    num_docs = len(featuresets)
    #to create 90-10 split
    train_set, test_set =
POS_featuresets[:math.floor(0.9*num_docs)],POS_featuresets[math.floor(0.9*num_docs):]
    classifier = nltk.NaiveBayesClassifier.train(train_set)
    print("Train set Accuracy: "+ str(nltk.classify.accuracy(classifier, train_set)))
    print("Test set Accuracy: "+ str(nltk.classify.accuracy(classifier, test_set)))
    # show most informative features
    print(classifier.show_most_informative_features(20))
    #kfold on SL feature
    print("******** K fold with SL features*******")
    cross_validation_accuracy(10,SL_featuresets)
    print("******** K fold with POS features*******")
    #Kfold on POS feature
    cross_validation_accuracy(10,POS_featuresets )
    #Kfold on doc feature
    print("******** K fold with document features*******")
    cross_validation_accuracy(10,featuresets )

    print("******** Kfold on document feature on multinomial naive bayes
classifier*******")
    #Kfold on document feature on multinomial naive bayes classifier
    cross_validation_accuracy_SKlearn(10,featuresets )

"""
commandline interface takes a directory name with semeval task b training subdirectory
       for downloaded-tweeti-b-dist.tsv
   and a limit to the number of tweets to use
It then processes the files and trains a tweet sentiment classifier.

"""
if __name__ == '__main__':
    if (len(sys.argv) != 3):
        print ('usage: classifytweets.py <corpus-dir> <limit>')
        sys.exit(0)
    processtweets(sys.argv[1], sys.argv[2])
```