REPORT - NLP (CSCE-689, Programming Assignment #4 Sentiment Lexicon Induction)
Name - Navneet Gupta
UIN - 226000691

1. Compile and Execution

It is developed with python 2.7 . The original training data(tagged and untagged data) and results are in the zip itself, along with the source code.

Steps -

1. Unzip the file.
2. cd into that folder
3. python python/SentimentAnalyzer.py --tagged=data/tagged --untagged=data/untagged
4. Output will get printed on the terminal

2. Results and Analysis

Results -

```
[INFO] Fold 0 Accuracy: 0.480000
[INFO] Fold 1 Accuracy: 0.505000
[INFO] Fold 2 Accuracy: 0.520000
[INFO] Fold 3 Accuracy: 0.520000
[INFO] Fold 4 Accuracy: 0.550000
[INFO] Fold 5 Accuracy: 0.535000
[INFO] Fold 6 Accuracy: 0.525000
[INFO] Fold 7 Accuracy: 0.505000
[INFO] Fold 8 Accuracy: 0.535000
[INFO] Fold 9 Accuracy: 0.550000
[INFO] Accuracy: 0.522500
```

3. Problems and Limitations -

a. Overlapping Regexes are not accounted for.
b. The training data is too small. Thus near operator gives results as 0 for many phrases.
c. The distribution of “great” and “poor” is very skewed in training data. “great” occurs >4 times as “poor”. This leads to very small values of semantic orientation and results in ‘neg’ classification most of the times.
d. Takes around 11 mins to run on my machine. This is because of calculating semantic orientation of all possible phrases in training phrase and this happens 10 times because of cross validation.

Key Steps Answers -

1. Commands to run the tagger

   A shell script(given below) to tag the data -

```
# Negative data
input_fol="/home/don/NLP/HW4/untagged/neg"
output_fol="/home/don/NLP/HW4/tagged/neg"
for filename in $input_fol/*
do
        fname=`basename $filename`
        ./tagchunk.i686 -predict . w-1 $filename ~/NLP/resources > $output_fol/$fname
done

# Positive data
input_fol="/home/don/NLP/HW4/untagged/pos"
output_fol="/home/don/NLP/HW4/tagged/pos"
for filename in $input_fol/*
do
        fname=`basename $filename`
        ./tagchunk.i686 -predict . w-1 $filename ~/NLP/resources > $output_fol/$fname
done
```

2. Regexes and Examples

   Regex ⇒ Example

   a. ([\S]+)_JJ_[A-Z-]+ ([\S]+)_NNS?_[A-Z-]+ ⇒ teen_JJ_I-NP couples_NNS_I-NP
   b. ([\S]+)_RB.?_[A-Z-]+ ([\S]+)_JJ_[A-Z-]+ ([\S]+)_((?!NN)..?.?)_[A-Z-]+ ⇒ pretty_RB_I-NP decent_JJ_I-NP teen_JJ_I-NP
   c. ([\S]+)_JJ_[A-Z-]+ ([\S]+)_JJ_[A-Z-]+ ([\S]+)_((?!NN)..?.?)_[A-Z-]+ ⇒ decent_JJ_I-NP teen_JJ_I-NP mind-fuck_JJ_I-NP
   d. ([\S]+)_NN.?_[A-Z-]+ ([\S]+)_JJ_[A-Z-]+ ([\S]+)_((?!NN)..?.?)_[A-Z-]+ ⇒ something_NN_B-NP other_JJ_B-ADJP than_IN_B-PP
   e. ([\S]+)_RB.?_[A-Z-]+ ([\S]+)_VB.?_[A-Z-]+ ⇒ even_RB_I-ADVP harder_VBD_B-VP

3. “Near” operator Code

```
def findPhrasePovertyAndGreatness(self, phrase, wordList):
    poor = 0
    great = 0
    for i in xrange(len(wordList) - 1):
        new_phrase = wordList[i] + " " + wordList[i + 1]
    if new_phrase == phrase:
        k = i + 11
        j = i + 2
        while j < len(wordList) and j <= k:
            if wordList[j] == 'poor':
                poor += 1
            elif wordList[j] == 'great':
                great += 1
            j += 1
        k = i - 10
        j = i - 1
        while j >= 0 and j >= k:
            if wordList[j] == 'poor':
                poor += 1
            elif wordList[j] == 'great':
                great += 1
            j -= 1
return poor, great
```

Returns “poor” and “great” counts NEAR that phrase.

4. Semantic Orientation of each sentiment phrase

```
great_count = self.greatDict[phrase]
poor_count = self.poorDict[phrase]
great = self.great
poor = self.poor
# Threshold condition
if great_count == 0 and poor_count == 0:
    continue
num = (great_count + 0.01) * (poor + 0.01)
den = (poor_count + 0.01) * (great + 0.01)
so = math.log(float(num) / den, 2)
```

Here “so” means the semantic orientation of the phrase.

5. Polarity Score of each test review

```
def classify(self, example):
    """
      Example is the test review to classify. Return 'pos' or 'neg' classification.
    """
    count = 0
    total_so = 0.0

    for regex in self.reg_exps:
        matches = re.findall(regex, example.taggedData)
        for m in matches:
            phrase = '%s %s' %(m[0], m[1])
            great_count = self.greatDict[phrase]
            poor_count = self.poorDict[phrase]
            great = self.great
            poor = self.poor
            # Threshold condition
            if great_count == 0 and poor_count == 0:
                continue
            num = (great_count + 0.01) * (poor + 0.01)
            den = (poor_count + 0.01) * (great + 0.01)
            so = math.log(float(num) / den, 2)
            total_so += so
            count += 1
    if count == 0:
        avg_so = 0.0
    else:
        avg_so = total_so / count
    if avg_so > 0:
        return 'pos'
    else:
        return 'neg'
```

Here example represents text of a review and in the end we return “pos”/”neg” for that.

Complete Source Code -

```
import sys
import getopt
import os
import math
import operator
from collections import defaultdict
import re

tagged_dir = ''
untagged_dir = ''
class TuringAlgo:
    class TrainSplit:
        """Represents a set of training/testing data. self.train is a list of Examples, as is self.test.
        """
        def __init__(self):
            self.train = []
            self.test = []

    class Example:
        """Represents a document with a label. klass is 'pos' or 'neg' by convention.
          words is a list of strings.
        """
        def __init__(self):
            self.taggedData = ''
            self.unTaggedData = ''
            self.klass = ''
            self.fileName = ''



    def __init__(self):
        """TuringAlgo initialization"""
        self.numFolds = 10
        self.greatDict = defaultdict(lambda : 0)
        self.poorDict = defaultdict(lambda : 0)
        self.poor = 0
        self.great = 0
        self.reg_exps = [
            '([\S]+)_JJ_[A-Z-]+ ([\S]+)_NNS?_[A-Z-]+',
            '([\S]+)_RB.?_[A-Z-]+ ([\S]+)_JJ_[A-Z-]+ ([\S]+)_((?!NN)..?.?)_[A-Z-]+',
            '([\S]+)_JJ_[A-Z-]+ ([\S]+)_JJ_[A-Z-]+ ([\S]+)_((?!NN)..?.?)_[A-Z-]+',
            '([\S]+)_NN.?_[A-Z-]+ ([\S]+)_JJ_[A-Z-]+ ([\S]+)_((?!NN)..?.?)_[A-Z-]+',
            '([\S]+)_RB.?_[A-Z-]+ ([\S]+)_VB.?_[A-Z-]+'
        ]


##################################################################################
#
    # TODO TODO TODO TODO TODO
    def classify(self, example):
        """ TODO
          'words' is a list of words to classify. Return 'pos' or 'neg' classification.
        """
        #print "classifying example : ", example.fileName
        count = 0
        total_so = 0.0
        # Write code here
        for regex in self.reg_exps:
            matches = re.findall(regex, example.taggedData)
            for m in matches:
                phrase = '%s %s' %(m[0], m[1])
                great_count = self.greatDict[phrase]
                poor_count = self.poorDict[phrase]
                great = self.great
                poor = self.poor
                # Threshold condition
                if great_count == 0 and poor_count == 0:
                    continue
                num = (great_count + 0.01) * (poor + 0.01)
                den = (poor_count + 0.01) * (great + 0.01)
                so = math.log(float(num) / den, 2)
                total_so += so
                count += 1
        if count == 0:
            avg_so = 0.0
        else:
            avg_so = total_so / count
        if avg_so > 0:
            return 'pos'
        else:
            return 'neg'

    def findPhrasePovertyAndGreatness(self, phrase, wordList):
        poor = 0
        great = 0
        for i in xrange(len(wordList) - 1):
            new_phrase = wordList[i] + " " + wordList[i + 1]
            if new_phrase == phrase:
                k = i + 11
                j = i + 2
                while j < len(wordList) and j <= k:
                    if wordList[j] == 'poor':
                        poor += 1
                    elif wordList[j] == 'great':
                        great += 1
                    j += 1
                k = i - 10
                j = i - 1
                while j >= 0 and j >= k:
                    if wordList[j] == 'poor':
                        poor += 1
                    elif wordList[j] == 'great':
                        great += 1
                    j -= 1
        return poor, great

    def findPovertyAndGreatness(self, wordList):
        poor = 0
        great = 0
        for word in wordList:
            if word == 'poor':
                poor += 1
            elif word == 'great':
                great += 1
        return (poor, great)

    def addExample(self, example):
        """
         * TODO
         * Train your model on an example document with label klass ('pos' or 'neg') and
         * words, a list of strings.
         * You should store whatever data structures you use for your classifier
         * in the TuringAlgo class.
         * Returns nothing
        """
        #print "adding example : ", example.fileName
        untaggedWordsList = re.split('\W+', example.unTaggedData)
        poor, great = self.findPovertyAndGreatness(untaggedWordsList)
```

```
        self.poor += poor
        self.great += great
        phrases_set = set()
        for regex in self.reg_exps:
            matches = re.findall(regex, example.taggedData)
            for m in matches:
                phrase = '%s %s'%(m[0], m[1])
                phrases_set.add(phrase)
        for phrase in phrases_set:
            poor, great = self.findPhrasePovertyAndGreatness(phrase, untaggedWordsList)
            if poor == 0 and great == 0:
                continue
            self.greatDict[phrase] += great
            self.poorDict[phrase] += poor
        # Write code here



    # END TODO (Modify code beyond here with caution)

#############################################################################
#



    def readFile(self, fileName):
        """
         * Code for reading a file.  you probably don't want to modify anything here,
         * unless you don't like the way we segment files.
        """
        with open(fileName, 'r') as myFile:
            data = myFile.read().replace('\n','')
        return data

    def crossValidationSplits(self, trainTaggedDir, trainUntaggedDir):
        """Returns a lsit of TrainSplits corresponding to the cross validation splits."""
        splits = []
        posTrainFileNames = os.listdir('%s/pos/' % trainTaggedDir)
        negTrainFileNames = os.listdir('%s/neg/' % trainTaggedDir)
        for fold in range(0, self.numFolds):
            split = self.TrainSplit()
            for fileName in posTrainFileNames:
                example = self.Example()
                example.taggedData = self.readFile('%s/pos/%s' % (trainTaggedDir, fileName))
                example.unTaggedData = self.readFile('%s/pos/%s' % (trainUntaggedDir, fileName))
                example.klass = 'pos'
                example.fileName = fileName

                if fileName[2] == str(fold):
                  split.test.append(example)
                else:
                  split.train.append(example)
            for fileName in negTrainFileNames:
                example = self.Example()
                example.taggedData = self.readFile('%s/neg/%s' % (trainTaggedDir, fileName))
                example.unTaggedData = self.readFile('%s/neg/%s' % (trainUntaggedDir, fileName))
                example.klass = 'neg'
                example.fileName = fileName

                if fileName[2] == str(fold):
                  split.test.append(example)
                else:
                  split.train.append(example)
            splits.append(split)
        return splits


def test10Fold(tagged_dir, untagged_dir):
    ta = TuringAlgo()
    splits = ta.crossValidationSplits(tagged_dir, untagged_dir)
    avgAccuracy = 0.0
    fold = 0
    for split in splits:
        classifier = TuringAlgo()
        accuracy = 0.0
        for example in split.train:
            classifier.addExample(example)

        for example in split.test:
            guess = classifier.classify(example)
            if example.klass == guess:
                accuracy += 1.0

        accuracy = accuracy / len(split.test)
        #print "split : ", accuracy
        avgAccuracy += accuracy
        print '[INFO]\tFold %d Accuracy: %f' % (fold, accuracy)
```

```
        fold += 1
    avgAccuracy = avgAccuracy / fold
    print '[INFO]\tAccuracy: %f' % avgAccuracy


def main():
    global tagged_dir
    global untagged_dir
    (options, args) = getopt.getopt(sys.argv[1:], [], ['tagged=', 'untagged='])
    for option, arg in options:
        if option == '--tagged':
            tagged_dir = arg
        elif option == '--untagged':
            untagged_dir = arg
    test10Fold(tagged_dir, untagged_dir)

if __name__ == "__main__":
    main()
```