Homework-3

Name: Priyank Devpura **UIN: 225002461**

Compile and Run:

a) **To compile and run Turney Algorithm: sentiment_main.py ./data/imdb1**

KEY STEP:

a) **POS Tagger command:**
   a. Used w-1 as weight. Developed a shell script to parse the all the files in one go. Check pos_run.sh and neg_run.sh script for positive and negative files.
   b. **Shell script:**

```
#!/bin/bash
echo "starting the run"
while read data_file
do
echo "current run is $data_file"
./tagchunk.i686 -predict . w-1 ./data/imdb1/pos/"$data_file" resources > ~/nlp_ass4/pos/"$data_file".out

done < pos_dat.txt
```

b) **Develop the procedures to parse through the files and check and get the pattern:**
   a. Identify_phrase procedure finds the indices of excellent and poor present in the document and then finds the list of words surrounding excellent or poor and then calls fill_dict procedure. The fill_dict fills the positive dictionary if the surrounding words are near to excellent. Similarly, it fills the negative dictionary.
   b. The search pattern is given in the form of list1, list2 and list3 as given in the paper.

```
self.list1 = [['JJ'], ['RB', 'RBR', 'RBS'], ['JJ'], ['NN', 'NNS'], ['RB', 'RBR', 'RBS']]
self.list2 = [['NN', 'NNS'], ['JJ'], ['JJ'], ['JJ'], ['VB', 'VBD', 'VBN', 'VBG']]
self.list3 = [[], ['NN', 'NNS'], ['NN', 'NNS'], ['NN', 'NNS'], []]
```

```
def fill_dict(self,word_list,word):
    # word_list = line.split()
    if word == 'excellent':
        dict=self.Posdict
    elif word == 'poor':
        dict=self.Negdict
    else:
        print "DO NOT COME HERE"
    word_list_len = len(word_list)
    for i in range(word_list_len - 1):
        each_word = word_list[i]
        split_word = each_word.split("_")
        split_word[0]=split_word[0].lower()
        found = 0
        # print split_word[1]
        for k in range(len(self.list1)):          //find the phrase corresponding to list1

            if split_word[1] in self.list1[k]:
```

```
            split_word2 = word_list[i + 1].split("_")
            split_word2[0]=split_word2[0].lower()
            for m in range(len(self.list2)):                    //find the phrase corresponding to list2
              if split_word2[1] in self.list2[m]:
                if (i == word_list_len - 2) or (m == 0) or (m == 4):
                  if (split_word[0], split_word2[0]) not in dict:
                    dict[(split_word[0], split_word2[0])] = 1
                    #print split_word[0], split_word2[0]
                  else:
                    dict[(split_word[0], split_word2[0])] += 1  //increase the count in the dictionary.
                    #print "never"
                  found = 1
                  break
                  # print split_word[0], split_word2[0]

                else:
                  split_word3 = word_list[i + 2].split("_")
                  if (split_word3[1] != 'NNS') and (split_word3[1] != 'NN'):
                    if (split_word[0], split_word2[0]) not in dict:
                      dict[(split_word[0], split_word2[0])] = 1
                    else:
                      dict[(split_word[0], split_word2[0])] += 1    //increase the count in the dictionary.
                      #print "never"
                    found = 1
                    break

            if found == 1:
              break
```

c) NEAR operator code: The indices corresponding to excellent and poor are found and then surrounding phrases are identified as shown in red below

```
def identify_phrase(self,data,word,word_list1,word_list2):
    data_len = len(data)
    #print word,data
    split_word=[]
    indices=[]
    for i,x in enumerate(data):
      split_word=x.split('_')
      split_word[0]=split_word[0].lower()
      if split_word[0]==word:
        indices.append(i)
        #print "indices: ",indices
    #indices = [i for i, x in enumerate(data) if x == word]

    if len(indices)>0:
      #print 'Found'
      for index in indices:
        if index <= 10 and data_len - index <= 10:
          word_list1 = data[:index]
          word_list2 = data[index:]
        elif index <= 10 and data_len - index >= 10:
          word_list1 = data[:index]
          word_list2 = data[index:index + 11]
        elif index >= 10 and data_len - index <= 10:
          word_list1 = data[index - 10:index]
          word_list2 = data[index:]
```

```
            else:
                word_list1 = data[index - 10:index]
                word_list2 = data[index:index + 11]
            if word == 'excellent':
                self.exe_hits +=1
            elif word == 'poor':
                self.poor_hits += 1
            else:
                print "DO nOT COME HERE"

            self.fill_dict(word_list1, word)
            self.fill_dict(word_list2, word)
```

d) and e) are done together. Relevant code to calculate Semantic Orientation and polarity:

a. The semantic orientation of the phrase is calculated using the formula given in paper.

b. The polarity is found by adding the semantic orientation of the relevant phrases present in the file.

```
def classify(self, word_list):
    """ TODO
      'words' is a list of words to classify. Return 'pos' or 'neg' classification.
    """

    word_list_len = len(word_list)
    so_val = 0.0
    for i in range(word_list_len - 1):
        num = 1.0
        denom = 1.0
        each_word = word_list[i]
        split_word = each_word.split("_")
        found = 0
        split_word[0]=split_word[0].lower()
        # print split_word[1]
        for k in range(len(self.list1)):
            is_pos=0
            is_neg=0
            if split_word[1] in self.list1[k]:
                split_word2 = word_list[i + 1].split("_")
                split_word2[0]=split_word2[0].lower()
                for m in range(len(self.list2)):
                    if split_word2[1] in self.list2[m]:
                        if (i == word_list_len - 2) or (m == 0) or (m == 4):
                            if (split_word[0], split_word2[0]) not in self.Posdict:

                                num = 0.01*self.poor_hits
                                is_pos=1
                            else:
                                num = self.Posdict[(split_word[0],split_word2[0])]*self.poor_hits
                            if (split_word[0], split_word2[0]) not in self.Negdict:
                                is_neg = 1
```

```
                denom = 0.01*self.exe_hits

                #print split_word[0], split_word2[0]
            else:
                denom = self.Negdict[(split_word[0],split_word2[0])]*self.exe_hits
            found = 1
            so_val+=self.find_log(num,denom)   //semantic orientation and polarity


            break
            # print split_word[0], split_word2[0]

        else:
            split_word3 = word_list[i + 2].split("_")
            if (split_word3[1] != 'NNS') and (split_word3[1] != 'NN'):
                if (split_word[0], split_word2[0]) not in self.Posdict:
                    # dict[(split_word[0], split_word2[0])] = 1
                    num = 0.01 * self.poor_hits
                    is_pos=1
                    #print split_word[0], split_word2[0]

                else:
                    num = self.Posdict[(split_word[0], split_word2[0])] * self.poor_hits
                if (split_word[0], split_word2[0]) not in self.Negdict:
                    # dict[(split_word[0], split_word2[0])] = 1
                    denom = 0.01 * self.exe_hits
                    is_neg=1
                    #print split_word[0], split_word2[0]
                else:
                    denom = self.Negdict[(split_word[0], split_word2[0])] * self.exe_hits
                if not (is_pos and is_neg):
                    so_val += self.find_log(num, denom)          // semantic orientation  and polarity

                found =1
                break


if so_val >= 0:
    return 'pos'
else:
    return 'neg'
```

RESULTS:

The average accuracy achieved using 10-fold validation is around 55.65. Detailed results are shown below in detailed results section

ANALYSIS:

A) The given dataset is not enough to do prediction with this algorithm as only some datasets contains excellent and poor and not all.
B) The phrases are not very much common between training set and test set.

C) If Semantic Orientation is derived by taking a bigger dataset as done in paper, we can get better results

LIMITATION:

A) The given dataset is limited and contains not many phrases that are common thus not providing a good accuracy.

DETAILED RESULTS:

[INFO] Fold 0 Accuracy: 0.575000
[INFO] Fold 1 Accuracy: 0.555000
[INFO] Fold 2 Accuracy: 0.575000
[INFO] Fold 3 Accuracy: 0.560000
[INFO] Fold 4 Accuracy: 0.520000
[INFO] Fold 5 Accuracy: 0.540000
[INFO] Fold 6 Accuracy: 0.620000
[INFO] Fold 7 Accuracy: 0.530000
[INFO] Fold 8 Accuracy: 0.520000
[INFO] Fold 9 Accuracy: 0.570000
[INFO] Accuracy: 0.556500