HOW TO COMPILE

Compile and run using Python 2. One command line argument for the file path to a directory containing separate directories for positive and negative reviews called “pos” and “neg”.

RESULTS AND ANALYSIS

Tagging Parts of Speech

```
for file in ~/CSCE689/sentimentAnalyzer/data/imdb1/pos/*
do
       ./tagchunk.i686 -predict . w-5-lc "$file" resources >
~/CSCE689/pa4/data/pos/${file##*/}.out
done

for file in ~/CSCE689/sentimentAnalyzer/data/imdb1/neg/*
do
       ./tagchunk.i686 -predict . w-5-lc "$file" resources >
~/CSCE689/pa4/data/neg/${file##*/}.out
done
```

To run tagchunk on all the files from the second programming assignment, I wrote a shell script that ran the program on every file in the “pos” folder using the w-5-lc weights file and then did the same thing on every file in the “neg” folder.

Generating Sentiment Phrases

```
def isValidPhrase(self, word1, word2, word3):
    phraseValid = False
    word1Pos = word1.split('_')[1]
    word2Pos = word2.split('_')[1]
    word3Pos = word3.split('_')[1]

    if word1Pos == 'JJ':
        if word2Pos == 'NN' or word2Pos == 'NNS':
            phraseValid = True
        elif word2Pos == 'JJ':
            if word3Pos != 'NN' and word3Pos != 'NNS':
                phraseValid = True
    elif word1Pos == 'RB' or word1Pos == 'RBR' or word1Pos == 'RBS':
        if word2Pos == 'JJ':
            if word3Pos != 'NN' and word3Pos != 'NNS':
                phraseValid = True
        elif word2Pos == 'VB' or word2Pos == 'VBD' or word2Pos == 'VBN' or
word2Pos == 'VBG':
            phraseValid = True
    elif word1Pos == 'NN' or word1Pos == 'NNS':
```

```
            if word2Pos == 'JJ':
                if word3Pos != 'NN' and word3Pos != 'NNS':
                    phraseValid = True

        return phraseValid
```

To generate the sentiment phrases iterated through the list of words in a review one word at a time passing each word as well as the two that followed it into a method that checked if the word was a valid phrase according to the criteria defined in the "Thumbs Up or Thumbs Down?" paper. I stopped two words short of the end of the words list. I don't think this method resulted in me missing any phrases because the last "word" according the the tagchunk tagger was always a punctuation mark, so the final two words could never be a valid phrase. Some examples of valid phrases from the tested reviews are "scene-stealing_JJ turn_NN", "immensely_RB entertaining_VBG", and "usually_RB annoying_JJ".

Implementing "NEAR" Operator

```
    def getHitsNear(self, index, nearWord, words):
        hits = 0
        lowerbound = 0 if index < 10 else (index - 10)
        upperbound = len(words) if (len(words) - index) < 11 else (index + 11)

        for i in range(lowerbound, upperbound):
            if words[i].split('_')[0] == nearWord:
                hits += 1

        return hits
```

To implement the near operator I wrote a method that took the index of the first word in a sentiment phrase, the nearWord I was checking was checking for ("excellent" or "poor") and the list of words in the review. I calculated a lower bound for the range by subtracting 10 from the index or setting the lower bound to 0 if the index was less than 10. I calculated an upper bound for the range by adding 11, to account for the second word in the phrase, or setting it equal to the length of the words list if the index was within 11 words of the end. Then I ran a for loop through the words list from the lower bound to the upper bound, checking to see if each word matched the nearWord, incrementing a counter every time there was a match and returning the counter at the end of the method.

Calculating Semantic Orientation

```
    def calculateSos(self):
        for phrase in self.phraseDict:
            numerator = (self.phraseDict[phrase]['hitsNearExcellent'] + 0.01) *
    self.hitsPoor
            denomenator = (self.phraseDict[phrase]['hitsNearPoor'] + 0.01) *
    self.hitsExcellent
            so = math.log(numerator/denomenator)
```

```
            self.phraseDict[phrase]['so'] = so
```

After running through all the reviews in the training set, I used the final hit counts for NEAR "excellent" and NEAR "poor" and the total hit counts for "excellent" and "poor" to calculate the semantic orientation for each phrase using equation 3 from the paper.

Calculating Polarity Score

```
    def classifyReview(self, words):
        soSum = 0.0
        klass = ''
        for i in range(0, len(words)-2):
            if self.isValidPhrase(words[i], words[i+1], words[i+2]):
                phrase = self.formPhrase(words[i], words[i+1])
                if phrase in self.phraseDict:
                    soSum += self.phraseDict[phrase]['so']

        if soSum < 0:
            klass = 'neg'
        else:
            klass = 'pos'

        return klass
```

To calculate the polarity score, I reused the isValidPhrase method to find phrases that matched the pattern described in the paper. If a valid phrase was in the dictionary, I added its semantic orientation value to a running total. Unlike the paper, I didn't take an average since I was just checking whether the value was positive or negative, not the magnitude of the positivity or negativity and taking an average would not change that value.

Results

```
[INFO]  Fold 0 Accuracy: 0.525000
[INFO]  Fold 1 Accuracy: 0.565000
[INFO]  Fold 2 Accuracy: 0.555000
[INFO]  Fold 3 Accuracy: 0.530000
[INFO]  Fold 4 Accuracy: 0.510000
[INFO]  Fold 5 Accuracy: 0.530000
[INFO]  Fold 6 Accuracy: 0.495000
[INFO]  Fold 7 Accuracy: 0.540000
[INFO]  Fold 8 Accuracy: 0.530000
[INFO]  Fold 9 Accuracy: 0.600000
[INFO]  Accuracy: 0.538000
```

Analysis

The program was an improvement over a random guess, but not significantly so with the most accurate fold achieving an accuracy of .600000 and the least accurate falling below .500000. This

could be due to a few factors. One possible reason is that, as noted in the paper, positive reviews can contain negative phrases, and vice versa. Another possible reason is that there weren't many occurrences of the words "excellent" or "poor" in the data set, weakening their predictive power.

KNOWN BUGS, PROBLEMS, AND LIMITATIONS

I'm not aware of any bugs or problems. As far as limitations, the program runs somewhat slowly, taking about 17 minutes to run through the imdb reviews and could probably be optimized, but it's not prohibitively slow.