Assignment 4: Programming Report

Lucas Machado

Student ID: 400507829

Scientific Programming: Rule-Based Sentiment Analysis

Sunday, April 7, 2024

1 Introduction

The program performs rule-based sentiment analysis, using the VADER lexicon rules to evaluate the sentiment of sentences. This report outlines the program's design, its functions, and usage instructions.

2 Text Files and Data

2.1 VADER Lexicon

The program uses text files; the first file (`vader_lexicon.txt`), which MIT created, contains the data. The file contains many words with specific stats about each word. Here is an example of a line in the text file:

```
clean 1.7 0.78102 [3, 1, 2, 1, 2, 1, 3, 2, 1, ...]
```

The data is in line with the word "clean," with a score of 1.7, a standard deviation of 0.78102, and an array of sentiment intensity scores represented by `[3, 1, 2, 1, 2, 1, 3, 2, 1, 1]`. The rest of the data in the text file has this same structure.

2.2 Validation Data

There is one more text file (`validation.txt`) that contains sentences. The program performs the sentiment analysis on the sentences in this file. Here is an example of how a sentence might be structured in the file:

```
VADER is not smart, handsome, nor funny.
```

All of the sentences in `validation.txt`, are analyzed by the program to determine sentiment scores based on previously mentioned lexicon data from `vader_lexicon.txt`.

3 Program Structure and Functions

There are three functions in the program, not including the main.

3.1 Function 1 readDictionaryAndPopulateWords

The primary purpose of the `readDictionaryAndPopulateWords` function is to get all the data from (`vader_lexicon.txt`) and add it to an array of words. The words are structs in C, meaning that we store the attributes of each word in it, such as the score and SD. This is done by reading each line in the text file, parsing the data for each word, and adding all the information to the specific word struct.

3.2 Function 2 computeSentimentScore

Now that we have all of the data in an array of words, we can use it to compute the sentiment score of a line of text, and this is what the function `computeSentimentScore` does. The function takes in a sentence, simply a string of characters. Then, the function will have a score (sentiment score) and a count for the number of words in the sentence. The function breaks up each sentence into words one at a time. It first looks at the first word in the sentence; it then iterates through our array of words and checks if the word is in the array. If the word is in the array, then it gets the score that is associated with that word, and it adds it to the score, but if it is not, then it adds 0 to the score. Then, at the end of the function, the total score will be divided by the number of words in the sentence. Thus, this gives the sentence score for a sentence. Note that this function operates on only one sentence (The one that is passed into the function).

3.3 performSentimentAnalysis

The primary purpose of the `performSentimentAnalysis` function is to print out the sentiment analysis performed on a sentence and call the `computeSentimentScore` to compute the score of each sentence. The function iterates through each line in the `validation.txt` and passes the sentence into `performSentimentAnalysis` to compute the sentence's score. Then, the function prints out the line that was read along with the sentiment score in an easy-to-read format.

4 Execution Instructions

First, run the command - `Make`
Second, run `./mySA vader_lexicon.txt validation.txt`

4.1 Compilation and Running

Here is an example of what the process of running the program in the terminal will look like:

```
make

gcc -Wall -g -o mySA mySA.c

user@hostname % ./mySA vader_lexicon.txt validation.txt


       string sample                                                                          score
-------------------------------------------------------------------------------------------------
VADER is smart, handsome, and funny.                                                          0.97
VADER is smart, handsome, and funny!                                                          0.97
VADER is very smart, handsome, and funny.                                                     0.83
VADER is VERY SMART, handsome, and FUNNY.                                                     0.83
VADER is VERY SMART, handsome, and FUNNY!!!                                                   0.83
VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY!!!                                      0.64
VADER is not smart, handsome, nor funny.                                                      0.83
The book was good.                                                                            0.47
At least it isn't a horrible book.                                                           -0.36
The book was only kind of good.                                                               0.61
The plot was good, but the characters are uncompelling and the dialog is not great.           0.27
Today SUX!                                                                                   -0.75
Today only kinda sux! But I'll get by, lol                                                    0.16
Make sure you :) or :D today!                                                                 0.80
Not bad at all                                                                               -0.62
```

5 Appendix

```
# Define compiler
CC = gcc

# Compiler flags
CFLAGS = -Wall -g

# Define the target executable
TARGET = mySA

# Default target
all: $(TARGET)

$(TARGET): $(TARGET).c
	$(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c

# Clean target for removing compiled objects and the executable
clean:
	$(RM) $(TARGET)
```

```
#ifndef FUNCTION_H
#define FUNCTION_H

// Function prototypes
struct words *readDictionaryAndPopulateWords(const char *filename, int *
    counterForNumberOfWordEntries);
```

```c
float computeSentimentScore(char *sentence, struct words *wordsList, int numOfWords);
void performSentimentAnalysis(const char *filename, struct words *wordsList, int numOfWords);

#endif // FUNCTION_H
```

```c
// Include the header file for the function prototypes
#include "functions.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Define a struct to store the word entries
struct words
{
    char *word;         // Pointer to store the word
    float score;        // Float to store the score
    float SD;           // Float to store the standard deviation
    int SIS_array[10]; // Array to store the sentiment intensity scores
};
/*
Function that reads the lines of the diconary file and populates the words struct, it into an array of words
*/
struct words *readDictionaryAndPopulateWords(const char *filename, int *counterForNumberOfWordEntries)
{

    // Open the file in read mode
    FILE *file = fopen(filename, "r");

    // Check if the file was opened without any issues
    if (!file)
    {
        // Print an error message if there's an issue opening the file
        printf("There was an error when opening the file please, check the file name or the path\n");
        // Return NULL if the file couldn't be opened
        return NULL;
    }

    // Initialize the initial size of the array of words, being able to store 300 characters for each line
    int capacityOfArray = 300;
    // Allocate memory for the array of words, with the capacity times the size of the struct
    struct words *wordsList = malloc(capacityOfArray * sizeof(struct words));

    // Check if memory allocation was successful, if not then return and error message
    if (!wordsList)
    {
        printf("The memory allocation for the array of words was not a success\n");
        // Close the file before returning
        fclose(file);
        // Return NULL due to failed memory allocation
        return NULL;
```

```
47      }
48
49      // Initialize the counter for the number of word entries to 0
50      *counterForNumberOfWordEntries = 0;
51      // Buffer to hold each line, it can hold up to 1024 characters
52      char line[1024];
53      while (fgets(line, sizeof(line), file))
54      {
55          // Check if we need to increase the capacity of the array
56          if (*counterForNumberOfWordEntries >= capacityOfArray)
57          {
58              // Double the capacity of the array
59              capacityOfArray *= 2;
60              // Reallocate memory for the array of words
61              struct words *tempEntries = realloc(wordsList, capacityOfArray *
                    sizeof(struct words));
62              // Check if memory reallocation was successful
63              if (!tempEntries)
64              {
65                  printf("The memory reallocation for the array of words was
                        not a success\n");
66                  // Free the memory allocated for the array of words
67                  free(wordsList);
68                  // Close the file before returning
69                  fclose(file);
70                  // return null meaning that the memeory reallocation was not
                        successful
71                  return NULL;
72              }
73              // Update the pointer to the new memory location
74              wordsList = tempEntries;
75          }
76
77          // Pointer to the current character in the line
78          char *currentCharacterPointer = line;
79          //// Track the current field word, score, SD, SIS_array being read
                in the current line
80          int currentFieldIndexInWords = 0;
81          // Temporary string to store the word, score, SD, SIS_array being
                read
82          char tempStr[256];
83          // Index to keep track of the current character in the temporary
                string
84          int tempStrIndex = 0;
85
86          // Loop through each character in the line
87          while (*currentCharacterPointer != '\0')
88          {
89              /*
90               Check if the current character is a tab open bracket comma
                    close bracket or a newline
91              */
92              if (*currentCharacterPointer == '\t' || *currentCharacterPointer
                    == '[' || *currentCharacterPointer == ',' ||
93                  *currentCharacterPointer == ']' || *currentCharacterPointer
                        == '\n')
94              {
95                  // Add a null terminator to the temporary string to make it
                        a full string in C
96                  tempStr[tempStrIndex] = '\0';
97
```

```
98                  // Pointer to the current word entry in the array
99                  struct words *currentWord = &wordsList[*
                        counterForNumberOfWordEntries];
100                 // Check the current field index to know which field we are
                        reading
101                 switch (currentFieldIndexInWords)
102                 {
103                 // Case for if the currentFieldIndexInWords is 0 meaning
                        that it is a word
104                 case 0:
105                     // This code is to allocate memory for the word and copy
                            the word into the memory
106                     // The strdup function allocates memory for the string
                        and copies the string into the memory
107                     currentWord->word = strdup(tempStr);
108                     break;
109                 // Case for if the currentFieldIndexInWords is 1 meaning
                        that it is a score
110                 case 1:
111                     // This code is to convert the score to a float and
                        store it in the struct
112                     // The atof function converts a string to a float and
                        allocates memory for the float
113                     // This is the same for case 2
114                     currentWord->score = atof(tempStr);
115                     break;
116                 // Case for if the currentFieldIndexInWords is 2 meaning
                        that it is a standard deviation
117                 case 2:
118                     currentWord->SD = atof(tempStr);
119                     break;
120                 // Case for if the currentFieldIndexInWords is 3-12 meaning
                        that it is a sentiment intensity score
121                 default:
122                     if (currentFieldIndexInWords - 3 < 10)
123                     {
124                         // Convert the sentiment intensity score to an
                            integer and store it in the struct
125                         currentWord->SIS_array[currentFieldIndexInWords - 3]
                                = atoi(tempStr);
126                     }
127                     break;
128                 }
129
130                 // Reset the index for the next field
131                 tempStrIndex = 0;
132                 // Move to the next field
133                 currentFieldIndexInWords++;
134
135                 // Check if the current character is a close bracket this
                        means that we are at the end of the array in the dict
                        file
136                 if (*currentCharacterPointer == ']')
137                 {
138                     // Increase the counter for the number of word entries
139                     *counterForNumberOfWordEntries += 1;
140                     currentFieldIndexInWords = 0; // Reset the field index
141                     break;
142                 }
143             }
144             // Check if the current character is not a space
```

```
145             else if (*currentCharacterPointer != ' ')
146             {
147                 // Storing the character pointed to by
                       currentCharacterPointer
148                 // in the tempStr array at the position tempStrIndex  and
                       then incrementing tempStrIndex by 1
149                 tempStr[tempStrIndex++] = *currentCharacterPointer;
150             }
151             // Move to the next character in the line
152             currentCharacterPointer++;
153         }
154     }
155 
156     // Close the file after reading all lines
157     fclose(file);
158     // Return the array of words
159     return wordsList;
160 }
161 
162 // Function to compute the sentiment score for a given sentence
163 float computeSentimentScore(char *sentence, struct words *wordsList, int
       numOfWords)
164 {
165     // Initialize the sum of scores and the word count
166     float sumScore = 0.0;
167     int wordCount = 0;
168 
169     // The strtok function is used to split the sentence into tokens
170     // this means that the sentence is split into words based on ,.!? and
           newline characters
171     // and it returns a pointer to the first token found in the sentence
172     char *token = strtok(sentence, " ,.!?\n");
173 
174     // Loop through each token until there are no more tokens
175     while (token != NULL)
176     {
177         // Convert the token to lowercase
178         // this has to be done becuase for the analysis we are looking at
               insensitive cases
179         for (int i = 0; token[i]; i++)
180         {
181             // Convert to lowercase
182             token[i] = tolower(token[i]);
183         }
184 
185         // Look up the token in the wordsList
186         int found = 0;
187         // This for loop is used to loop through the wordsList to find the
               token
188         for (int i = 0; i < numOfWords; i++)
189         {
190             // Check if the token is found in the wordsList
191             if (strcmp(wordsList[i].word, token) == 0)
192             {
193                 // Add the score of the word to the sum of scores
194                 sumScore += wordsList[i].score;
195                 // Set found to 1 to indicate that the word was found
196                 found = 1;
197                 break;
198             }
199         }
```

```
200         // If the word is not found in the wordsList then the score is 0
201         if (!found)
202         {
203             sumScore += 0; // Add zero to the sum if not found becuase the
                    word does not mean anything such as the word "lucas"
204         }
205         // Increment the word count
206         wordCount++;
207 
208         // Get the next token
209         token = strtok(NULL, " ,.!?\n");
210     }
211 
212     // Check if any words were found
213     if (wordCount > 0)
214     {
215         // Return the average score of the sentence
216         return sumScore / wordCount;
217     }
218     else
219     {
220         // Return 0 if no words were found
221         return 0;
222     }
223 }
224 
225 // Function to perform sentiment analysis on sentences from a file
226 void performSentimentAnalysis(const char *filename, struct words *wordsList,
        int numOfWords)
227 {
228     // Open the file in read mode
229     FILE *file = fopen(filename, "r");
230     // Check if the file was opened successfully
231     if (!file)
232     {
233         perror("Error opening sentences file");
234         return;
235     }
236 
237     // Buffer to hold each line it can hold at most 450 characters
238     char line[450];
239 
240     // Print the header for the output the -85s is used to left align the
           string
241     printf("%-85s %s\n", "       string sample", "score");
242     printf("
           ----------------------------------------------------------------------------------
           n");
243 
244     // Loop through each line in the file
245     while (fgets(line, sizeof(line), file) != NULL)
246     {
247         if (line[strlen(line) - 1] != '\n' && !feof(file)) // Check if the
               line is too long for the buffer
248         {
249             // Handle the error for the line exceeding buffer size
250             fprintf(stderr, "Line too long for buffer and was shorted please
                    keep note there may be issues.\n");
251 
252             // skip the rest of the line that didn't fit into the buffer
253             int ch;
```

```
254             while ((ch = fgetc(file)) != '\n' && ch != EOF)
255                 ;
256             // Continue to the next line
257             continue;
258         }
259 
260         // Remove the newline character from the line
261         line[strcspn(line, "\r\n")] = 0;
262 
263         // Buffer to store the original line before tokenization
264         char originalLine[450];
265         // Copy the line
266         strcpy(originalLine, line);
267 
268         // Get the sentiment score for the line
269         float score = computeSentimentScore(line, wordsList, numOfWords);
270 
271         // Print the original line and the score
272         printf("%-85s %.2f\n", originalLine, score);
273     }
274 
275     fclose(file); // Close the file
276 }
277 
278 int main(int argc, char *argv[])
279 {
280     if (argc != 3)
281     { // Ensure correct number of arguments
282         printf("Usage: %s <dictionary_filename> <sentences_filename>\n",
                argv[0]);
283         return 1;
284     }
285 
286     // Read the dictionary file and populate the words struct
287     int counterForTheNumberOfEntriesInTheFile = 0;
288     struct words *lexiconEntries = readDictionaryAndPopulateWords(argv[1], &
        counterForTheNumberOfEntriesInTheFile);
289 
290     // Check if there was an issue reading the lexicon or the lexicon is
        empty
291     if (!lexiconEntries || counterForTheNumberOfEntriesInTheFile == 0)
292     {
293         printf("There was an error in reading the lexicon or the file.\n");
294         return 1;
295     }
296     // Perform sentiment analysis on the sentences from the file
297     performSentimentAnalysis(argv[2], lexiconEntries,
        counterForTheNumberOfEntriesInTheFile);
298 
299     // Free the memory allocated for the words
300     for (int i = 0; i < counterForTheNumberOfEntriesInTheFile; i++)
301     {
302         free(lexiconEntries[i].word); // Free each word
303     }
304     // Free the memory allocated for the array of words
305     free(lexiconEntries);
306 
307     return 0;
308 }
```