Rule-Based Sentiment Analysis

George Ghiugan

April 10, 2024

Description:

This C program uses rule-based sentiment analysis technique to analyze the sentiment of given sentences. Using data such as sentiment scores of various words given in vader_lexicon.txt, the program computes the average sentiment scores of a given sentence and displays it to the console. The use of sentiment analysis is useful in analyzing the overall sentiment of given text. The applications of Sentiment Analysis allow companies to improve product/services in a variety of fields. Nowadays with the integration of Artificial Intelligence, the use of AI-Sentiment Analysis has become important in parsing large sets of data. The program developed in this assignment is a C implementation of how Sentiment Analysis works to give an understanding towards this important topic.

Code Analysis:

main.c:

The main function runs the program by reading in the lexicon file and an input file provided as command-line arguments. It then calls appropriate functions to calculate the sentiment score average for a sentence and displays it to the console:

```c
int main(int argc, char *argv[]) {

    // Error message for invalid number of parameter inputs
    if (argc != 3) {
        // Print the usage of how to correctly compile the program
        fprintf(stderr, "Usage: %s <vader_lexicon.txt> <input_file.
           txt>\n", argv[0]);
        return 1;
    }

    // Declare structure for lexicon and dimension
    struct words *lexicon;
    int lexicon_dimension;

    // Read the lexicon file and populate the structure
    if (!read_vader_file(argv[1], &lexicon, &lexicon_dimension)) {
        return 1;
    }

    // open the input file in read mode
    FILE *input_file = fopen(argv[2], "r");
    // Check if file can be read
    if (!input_file) {
        // Print error message
        fprintf(stderr, "Error opening file %s\n", argv[2]);
        return 1;
    }

    // Format print for the header on the console
    printf("   string sample                            score\n");

    // Set size for line variable
    char line[256];

    // Read each line of input file
    while (fgets(line, sizeof(line), input_file)) {
        // Remove newline character from the end of the line
        line[strcspn(line, "\n")] = 0;
        // Calculate the average sentiment score and print to
           console with formatting
        float score = calculate_sentiment_score(line, lexicon,
           lexicon_dimension);
        printf("%-85s %20.2f\n", line, score);
    }

    // Close the file
    fclose(input_file);

    // Free the memory for lexicon structure as well as for lexicon
        dimension
    free_lexicon(lexicon, lexicon_dimension);

    return 0;
}
```

vader_lexicon.c:

This function is responsible for parsing the contents of vader_lexicon.txt and storing the data in the appropriate data structure defined. The function also computes the average sentiment score of given sentences in the validation.txt file.

Header/Library files: Include the .h file and all the necessary standard libraries to be used in the program. Also define the maximum length for line and words.

```
#include "vader_lexicon.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Defining maximum length for line to be parsed and for the word
#define MAX_LINE_LENGTH 256
#define MAX_WORD_LENGTH 50
```

int read_vader_file(const char *filename, struct words **lexicon, int *lexicon_dimension):

This function reads the "vader_lexicon.txt" file and stores its data within the defined words structure. This is how it works:

Reading File: Opening the input file in read mode and making sure it is successful:

```
// Opening the file in reading mode
    FILE *file = fopen(filename, "r");
    // Check if opening the file was unsucessful
    if (!file) {
        // Print Error message indicating file opening was not
```

```
           sucessful
        fprintf(stderr, "Error opening file %s\n", filename);

        // Return 0 indicating failure in opening file
        return 0;
    }

    // Declare and initialize pointers for array size and memory
       allocation.
    *lexicon_dimension = 0;
    *lexicon = NULL;

    // Set the size of the line length to be parsed
    char line[MAX_LINE_LENGTH];
```

Parsing the Data: In the while loop each line from the file is read using 'fgets()'. For each line being read, the data is parsed to retrieve the word, score, standard deviation (SD), and the array of sentiment intensity scores (SIS). If the parsing is successful, memory is dynamically allocated for a new entry in the lexicon structure with data such as the word, score, SD, and SIS_array stored in the allocated memory.

```
while (fgets(line, sizeof(line), file)) {

        // Set the size of the max word length
        char word[MAX_WORD_LENGTH];

        // Declare variables for data to be stored from file
        float score, SD;
        int SIS_array[10];

        // Parse the line and retrieve the word, score, SD, and
           SIS_array
        if (sscanf(line, "%s %f %f [%d, %d, %d, %d, %d, %d, %d, %d,
            %d, %d]",
                   word, &score, &SD,
                   &SIS_array[0], &SIS_array[1], &SIS_array[2], &
                      SIS_array[3],
                   &SIS_array[4], &SIS_array[5], &SIS_array[6], &
                      SIS_array[7],
                &SIS_array[8], &SIS_array[9]) != 13) {
            continue;
        }

        // Allocate memory for the new lexicon entry
        struct words *temp = (struct words *)realloc(*lexicon, (*
          lexicon_dimension + 1) * sizeof(struct words));
        // Check if memory allocation was successful
        if (!temp) {
            // Print Error message if unsuccessful, close the file,
               and return 0
            fprintf(stderr, "Memory allocation error\n");
            fclose(file);
            return 0;
        }

        // Update the lexicon pointer
        *lexicon = temp;

        // Allocate memory for the word
        (*lexicon)[*lexicon_dimension].word = (char *)malloc((
          strlen(word) + 1) * sizeof(char));
        // Check if memory allocation is successful
        if (!((*lexicon)[*lexicon_dimension].word)) {
            // Print Error message if unsuccessful, close the file,
               and return 0
            fprintf(stderr, "Memory allocation error\n");
            fclose(file);
            return 0;
        }
        // Copy the word to the lexicon entry
        strcpy((*lexicon)[*lexicon_dimension].word, word);

        // Assign the score and Standard Deviation value to lexicon
            entry
        (*lexicon)[*lexicon_dimension].score = score;
        (*lexicon)[*lexicon_dimension].SD = SD;

        // Loop in order to assign values to the SIS array in the
```

```
            lexicon entry
        for (int i = 0; i < 10; i++) {
            (*lexicon)[*lexicon_dimension].SIS_array[i] = SIS_array
               [i];
        }

        //Increment the size of lexicon
        (*lexicon_dimension)++;
    }
```

Closing File: After parsing all the lines of the text file, in this step, the file is being closed and return a value of 1 indicating successful parsing of the vader_lexicon.txt:

```
    // Close file and return value of 1
    fclose(file);
    return 1;
```

float calculate_sentiment_score(const char *sentence, struct words *lexicon, int lexicon_dimension):

This function calculates the average sentiment score for a given sentence from validation.txt and returns the value as a float. Here is how it works:

Tokenization: Here two variables are declare and initialize which will keep track of score and word count when parsing the sentences. We then create a duplicate of the sentence stored in `*processed_sentence` and then **tokenize** it using white-space as the delimiter and storing it in `*token`:

```
    // Declare and initialize variables for tracking score and word
        count
    float total_score = 0.0;
    int word_count = 0;

    // Create a duplicate of the input sentence
    char *processed_sentence = strdup(sentence);
    // Check if memory allocation is successful
    if (!processed_sentence) {
         // Print error message if unsuccessful and return 0.0
        fprintf(stderr, "Memory allocation error\n");
        return 0.0;
    }

    // Tokenizes the sentence using whitespace as the delimiter
    char *token = strtok(processed_sentence, " ");
```

Processing Token and Compare: After tokenizing the words from the sentence the while loop iterates through each word, removes any unnecessary punctuation while preserving special characters, and compares the word to the lexicon to see if it matches. If there is a match, then the total_score variable adds the respective score from the lexicon structure as well as the word count. If no match, the word count still increments to correctly calculate the total average of sentiment score. Note that some special characters are preserved due to the special characters being part of certain words in vader_lexicon.txt:

```
    // Iterate through each of the tokens retrieved
    while (token != NULL) {
        // Remove punctuation, but preserve special characters
           within the text file
        int j = 0;
        // Iterate through each character in the token
        for (int i = 0; token[i]; i++) {
            char c = token[i];
            // Preserve specific characters while removing
               punctuation
            if ( isalnum(c) || c == '\'' || c == '(' || c == ')' ||
                c == '%' || c == '^' || c == '>' || c == '-' ||
                c == '<' || c == '/' || c == '*' || c == '@' || c
                   == ':' || c == '{' || c == '}' ||
                c == '$' || c == '\\' || c == '#') {
                // Copy valid characters to the processed_sentence
                processed_sentence[j++] = c;
            }
        }
        // Null-terminate the processed_sentence
        processed_sentence[j] = '\0';

        // Convert token to lowercase
        for (int i = 0; processed_sentence[i]; i++) {
            processed_sentence[i] = tolower(processed_sentence[i]);
        }


        int found = 0;

        // Iterate and search for the token in the lexicon
           structure
        for (int i = 0; i < lexicon_dimension; i++) {
            // Compared the token and see if it matches with any
               word in the lexicon
            if (strcmp(processed_sentence, lexicon[i].word) == 0) {
                // Increment the score and word count for match
                total_score += lexicon[i].score;
                word_count++;
                found = 1;
                break;
            }
        }

        // Check if the token is not found in the lexicon
        if (!found) {
            // Assign a score of 0 for unknown words and increment
               count
            total_score += 0;
            word_count++;
        }


        // Move to the next token
        token = strtok(NULL, " ");
    }
```

Free Memory and Return Value: Finally, we have to free the memory from `processed_sentence` and return the calculated average sentiment score for the sentence parsed:

```
    // Free the memory allocated to processed_sentence
    free(processed_sentence);

    // Calculate Sentiment score and return it
    return word_count > 0 ? total_score / word_count : 0.0;
```

Free Memory lexicon: We now free the memory allocated for the lexicon. We iterate through each word and free the memory for each word individually and then we free the memory for the entire lexicon structure to prevent memory leaks:

```
// Function to free memory allocated for the lexicon
void free_lexicon(struct words *lexicon, int lexicon_dimension) {
    // Loop through each word
    for (int i = 0; i < lexicon_dimension; i++) {
        // Free the memory for each word
        free(lexicon[i].word);
    }
    // Free the memory for the structure
    free(lexicon);
}
```

Results:

The results of the program is displayed in the console which shows each sentence with its corresponding average sentiment score. To run the program use the command:

`./mySA vader_lexicon.txt validation.txt`

These are the results of the average sentiment scores given the validation.txt file:

```
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
```

Note: Run the program on your computer to see proper formatting of scores and header aligned

Overall, the sentiment scores calculated in this program in accordance with the vader_lexicon.txt predefined values provide a quantitative measure of the emotional tone or sentiment conveyed by each sentence analyzed. Using data from a lexicon file this can help to analyze and understand the sentiment expressed in the given textual data as seen through this program.

Valgrind Check:

To ensure that the memory was properly freed and that there were no memory leaks, the valgrind tool was used to produce a detailed summary. The following command was used:

`valgrind -leak-check=full ./mySA vader_lexicon.txt validation.txt`:

```
==323439== HEAP SUMMARY:
==323439==     in use at exit: 0 bytes in 0 blocks
==323439==   total heap usage: 15,053 allocs, 15,053 frees,
   1,582,019,134 bytes allocated
==323439==
==323439== All heap blocks were freed -- no leaks are possible
==323439==
==323439== For lists of detected and suppressed errors, rerun with:
    -s
==323439== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
   from 0)
}
```

The "in use at exit: 0 bytes in 0 blocks": indicates that no memory was in use (allocated) at the time of program exit. This suggests that all allocated memory was properly deallocated (freed) before the program terminated.The summary overall shows that there are 0 errors from 0 contexts meaning that the memory has been properly freed within the program.

Appendix:

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "vader_lexicon.h"

// Main function which runs the program and displays results to the
    console
int main(int argc, char *argv[]) {

    // Error message for invalid number of parameter inputs
    if (argc != 3) {
        // Print the usage of how to correctly compile the program
```

```c
        fprintf(stderr, "Usage: %s <vader_lexicon.txt> <input_file.
           txt>\n", argv[0]);
        return 1;
    }

    // Declare structure for lexicon and dimension
    struct words *lexicon;
    int lexicon_dimension;

    // Read the lexicon file and populate the structure
    if (!read_vader_file(argv[1], &lexicon, &lexicon_dimension)) {
        return 1;
    }

    // open the input file in read mode
    FILE *input_file = fopen(argv[2], "r");
    // Check if file can be read
    if (!input_file) {
        // Print error message
        fprintf(stderr, "Error opening file %s\n", argv[2]);
        return 1;
    }

    // Format print for the header on the console
    printf("   string sample                             score\n");


    // Set size for line variable
    char line[256];

    // Read each line of input file
    while (fgets(line, sizeof(line), input_file)) {
        // Remove newline character from the end of the line
        line[strcspn(line, "\n")] = 0;
        // Calculate the average sentiment score and print to
           console with formatting
        float score = calculate_sentiment_score(line, lexicon,
           lexicon_dimension);
        printf("%-85s %20.2f\n", line, score);
    }

    // Close the file
    fclose(input_file);

    // Free the memory for lexicon structure as well as for lexicon
        dimension
    free_lexicon(lexicon, lexicon_dimension);

    return 0;
}
```

vader_lexicon.c:

```
#include "vader_lexicon.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Defining maximum length for line to be parsed and for the word
#define MAX_LINE_LENGTH 256
#define MAX_WORD_LENGTH 50

// Function which reads the "vader_lexicon.txt" file and stores its
    data within the defined "words" structure.
int read_vader_file(const char *filename, struct words **lexicon,
   int *lexicon_dimension) {

    // Opening the file in reading mode
    FILE *file = fopen(filename, "r");
    // Check if opening the file was unsucessful
    if (!file) {
        // Print Error message indicating file opening was not
           sucessful
        fprintf(stderr, "Error opening file %s\n", filename);

        // Return 0 indicating failure in opening file
        return 0;
    }

    // Declare and initialize pointers for array size and memory
       allocation.
    *lexicon_dimension = 0;
    *lexicon = NULL;

    // Set the size of the line length to be parsed
    char line[MAX_LINE_LENGTH];

    //  Reading each line from the file
    while (fgets(line, sizeof(line), file)) {

        // Set the size of the max word length
        char word[MAX_WORD_LENGTH];

        // Declare variables for data to be stored from file
        float score, SD;
        int SIS_array[10];

        // Parse the line and retrieve the word, score, SD, and
           SIS_array
        if (sscanf(line, "%s %f %f [%d, %d, %d, %d, %d, %d, %d, %d,
            %d, %d]",
                   word, &score, &SD,
                   &SIS_array[0], &SIS_array[1], &SIS_array[2], &
                      SIS_array[3],
                   &SIS_array[4], &SIS_array[5], &SIS_array[6], &
                      SIS_array[7],
                   &SIS_array[8], &SIS_array[9]) != 13) {
            continue;
        }

        // Allocate memory for the new lexicon entry
        struct words *temp = (struct words *)realloc(*lexicon, (*
           lexicon_dimension + 1) * sizeof(struct words));
        // Check if memory allocation was successful
        if (!temp) {
            // Print Error message if unsuccessful, close the file,
                and return 0
            fprintf(stderr, "Memory allocation error\n");
            fclose(file);
            return 0;
        }

        // Update the lexicon pointer
        *lexicon = temp;

        // Allocate memory for the word
        (*lexicon)[*lexicon_dimension].word = (char *)malloc((
           strlen(word) + 1) * sizeof(char));
        // Check if memory allocation is successful
        if (!((*lexicon)[*lexicon_dimension].word)) {
            // Print Error message if unsuccessful, close the file,
                and return 0
            fprintf(stderr, "Memory allocation error\n");
            fclose(file);
            return 0;
        }
        // Copy the word to the lexicon entry
        strcpy((*lexicon)[*lexicon_dimension].word, word);

        // Assign the score and Standard Deviation value to lexicon
            entry
        (*lexicon)[*lexicon_dimension].score = score;
        (*lexicon)[*lexicon_dimension].SD = SD;

        // Loop in order to assign values to the SIS array in the
           lexicon entry
        for (int i = 0; i < 10; i++) {
            (*lexicon)[*lexicon_dimension].SIS_array[i] = SIS_array
               [i];
        }

        //Increment the size of lexicon
        (*lexicon_dimension)++;
    }
    // Close file and return value of 1
    fclose(file);
    return 1;
}

// Function which calculates the sentiment score average
float calculate_sentiment_score(const char *sentence, struct words
   *lexicon, int lexicon_dimension) {

    // Declare and initialize variables for tracking score and word
       count
    float total_score = 0.0;
    int word_count = 0;

    // Create a duplicate of the input sentence
    char *processed_sentence = strdup(sentence);
    // Check if memory allocation is successful
    if (!processed_sentence) {
         // Print error message if unsuccessful and return 0.0
        fprintf(stderr, "Memory allocation error\n");
        return 0.0;
    }

    // Tokenizes the sentence using whitespace as the delimiter
    char *token = strtok(processed_sentence, " ");
    // Iterate through each of the tokens retrieved
    while (token != NULL) {
        // Remove punctuation, but preserve special characters
           within the text file
        int j = 0;
        // Iterate through each character in the token
        for (int i = 0; token[i]; i++) {
            char c = token[i];
            // Preserve specific characters while removing
               punctuation
            if ( isalnum(c) || c == '\'' || c == '(' || c == ')' ||
                c == '%' || c == '^' || c == '>' || c == '_' ||
                c == '<' || c == '/' || c == '*' || c == '@' || c
                   == ':' || c == '{' || c == '}' ||
                c == '$' || c == '\\' || c == '#') {
                // Copy valid characters to the processed_sentence
                processed_sentence[j++] = c;
            }
        }
        // Null-terminate the processed_sentence
        processed_sentence[j] = '\0';

        // Convert token to lowercase
        for (int i = 0; processed_sentence[i]; i++) {
            processed_sentence[i] = tolower(processed_sentence[i]);
        }


        int found = 0;

        // Iterate and search for the token in the lexicon
           structure
        for (int i = 0; i < lexicon_dimension; i++) {
            // Compared the token and see if it matches with any
               word in the lexicon
            if (strcmp(processed_sentence, lexicon[i].word) == 0) {
                // Increment the score and word count for match
                total_score += lexicon[i].score;
                word_count++;
                found = 1;
                break;
            }
        }

        // Check if the token is not found in the lexicon
        if (!found) {
            // Assign a score of 0 for unknown words and increment
               count
            total_score += 0;
            word_count++;
        }
```

```
        // Move to the next token
        token = strtok(NULL, " ");
    }


        // Free the memory allocated to processed_sentence
        free(processed_sentence);

        // Calculate Sentiment score and return it
        return word_count > 0 ? total_score / word_count : 0.0;
}

// Function to free memory allocated for the lexicon
void free_lexicon(struct words *lexicon, int lexicon_dimension) {
    // Loop through each word
    for (int i = 0; i < lexicon_dimension; i++) {
        // Free the memory for each word
        free(lexicon[i].word);
    }
    // Free the memory for the structure
    free(lexicon);
}
```

vader_lexicon.h:

```
#ifndef VADER_LEXICON_H
#define VADER_LEXICON_H
// Libraries for standard input/output and strings.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure to store data for sentiment of words
struct words {
    char *word;
    float score;
    float SD;
    int SIS_array[10];
};
```

```
// Function that parses the vader_lexicon.txt and populates the
   words structure
int read_vader_file(const char *filename, struct words **lexicon,
   int *lexicon_size);

// Function which tokenizes words from validation.txt and
   calculates the average sentiment score for the sentences
float calculate_sentiment_score(const char *sentence, struct words
   *lexicon, int lexicon_size);

// Free dynamically allocated memory for lexicon
void free_lexicon(struct words *lexicon, int lexicon_size);

#endif /* VADER_LEXICON_H */
```

Makefile:

```
CC = gcc
CFLAGS = -Wall -Wextra -lm

all: main

main: main.c vader_lexicon.c vader_lexicon.h
 $(CC) -o mySA main.c vader_lexicon.c $(CFLAGS)

clean:
 rm -f mySA
```