Assignment 3: Implementing VADER Sentiment Analysis in C

Talha Ahmad, 400517273

November 14, 2024

1 Introduction

In this assignment, we were tasked with designing an implementing a VADER Sentiment Analysis program in C. VADER is a lexicon and rule-based sentiment analysis tool that is widely used in the field of Natural Language Processing (NLP). I will discuss my algorithm, as well as compare it with the Python implementation of VADER at the end of this report.

2 Algorithm

Our algorithm is divided in 2 main overlying portions: the reading of the data file and the sentiment analysis. Both are further split into smaller functions to make the code more modular and easier to understand, and we will discuss these steps in detail.

2.1 Reading the Data File

Reading the data file is the first step in our algorithm. We open the file and allocate an array of type `WordData` to store the contents of the file. Ensuring that memory was allocated successfully, we then read the word, value1, and value2 using the `fscanf` function. If the file has reached the end, which is indicated by the return value of `feof`, we break out of the loop. We then get the remainder of the line with the `fgets` function, and store the integer values in an array associated with the `WordData` struct. Finally, we store the `WordData` struct in an array of `WordData` structs. We also dynamically allocate memory for the `WordData` array, and free it at the end of the program. After closing the file, the pointer to the array is returned and the function has reached its end.

2.2 Sentiment Analysis

The sentiment analysis portion of the program is the most important part of the algorithm. The function takes a sentence and the data file as arguments, and returns the compound sentiment value of the sentence. It is handled in multiple steps, each of which is discussed below.

2.2.1 Discussion of Variables

A few variables are used in the sentiment analysis portion of the program. There is an index variable to store the index of the current word in the array; a sum variable to store the sum of the sentiment values of the words; and a count variable to store the number of words in the sentence. There are also 2 variables to store the intensifier and negation (in the event of an all caps case) values should such an operation be performed on the word. In each iteration, there is a boolean variable to check if the word is in all caps.

2.2.2 Tokenizing the Sentence

The first step in the sentiment analysis is to tokenize the sentence. This is done as the program computes the sentiment value of each word in the sentence. We start by tokenizing the sentence using the `strtok` function, and store the token in a variable. At the same time, there is a for loop to check if the word is in all caps, and convert it to lower case if that is the case, and also to check if there are any exclamation marks which are limited to 3.

2.2.3 Finding the Word in the Data File

The next step is to find the word in the data file. This is done in a separate function, which takes the word and the data file as arguments. The function linearly searches the array of `WordData` structs for the word, and returns the WordData struct if the word is found. Otherwise, it returns a `WordData` struct with the word set to the null character.

2.2.4 If Word is Found

If the word is found in the data file, the program proceeds to calculate the sentiment value of the word. This is done by first adding whatever value1 is to the sum variable. If the word is in all caps, we multiply the sum by the caps constant, which is 1.5. Then, if there is an existing intensifier or negation value, we first add the intensifier value to the sum, and then multiply the sum by the negation value. Finally, if this word had exclamation marks, we add the exclamation score to the sum if the sum is positive, and subtract it if the sum is negative. This is because if the word is "good!!!" it represents a postive sentiment whereas "bad!!!" represents a negative sentiment.

2.2.5 If Word is Not Found

If the word is not found, then we need to check if it is an intensifier or negation. This is done by iterating through both intensifier arrays and negation array, and checking if the word is in any of them. If it is, the value of the earlier defined intensifier or negation variable is set to the value of the intensifier or negation. Moreover, if the word is in all caps, we multiply the variables by the caps constant, which is 1.5.

The intensifier value is reset to 0 once it is used, but the negation value is not reset until the next negation word is found. This is because if we consider a sentence like "good, very cool, and happy", the intensifier value is declared when the token "very" is found, but then it's used up for "cool" and reset to 0 so the other words don't get affected. However, if we

consider a sentence like "not good, happy, and excited", the negation value says that, in this context, all the words are negative, so it's not reset until the next negation word is found.

After the sentiment value of the word is calculated, the program proceeds to the next word in the sentence. At the end, the program returns the compound sentiment value of the sentence with the following formula:

$$\text{Compound Sentiment} = \frac{\text{Sum}}{\sqrt{\text{Sum}^2 + 15}}$$

For the pos, neg, and neu scores, I summed up the number of positive, negative, and neutral value words in a sentence and then normalized them by dividing by the total sum of the scores. This gives me a percentage of the positive, negative, and neutral words in the sentence:

$$\text{Positive} = \frac{\text{Positive Score}}{\text{Total Score}}$$

$$\text{Negative} = \frac{\text{Negative Score}}{\text{Total Score}}$$

$$\text{Neutral} = \frac{\text{Neutral Score}}{\text{Total Score}}$$

3 Header File

In the `utility.h` header file, I defined the `WordData` struct, as well as the function prototypes for the functions in the program. I also defined the intensifier, exclamation, and negation constants in the header file. The list of positive and negative intensifiers and negation words are also defined in the header file. This was managed through declaring the array as a static char array so it wouldn't be declared multiple times in the program.

4 Compiling the Program

To compile the program, the Makefile can be used:

```
make
```

This will compile the program and create an executable called `vader`. This executable can be run with the following command:

```
./vader
```

5 Python Implementation and Comparison

The Python implementation was done using the `NLTK` library, which has a built-in VADER Sentiment Analysis tool. The following table shows the comparison between the C and Python implementations:

Sentence	C Sentiment	Python Sentiment
VADER is smart, handsome, and funny.	0.8316	0.8316
VADER is smart, handsome, and funny!	0.854	0.844
VADER is very smart, handsome, and funny.	0.852	0.8545
VADER is VERY SMART, handsome, and FUNNY.	0.914	0.9227
VADER is VERY SMART, handsome, and FUNNY!!!	0.945	0.9342
VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY!!!	0.945	0.9469
VADER is not smart, handsome, nor funny.	-0.599	-0.7424
At least it isn't a horrible book.	0.307	0.431
The plot was good, but the characters are uncompelling and the dialog is not great.	-0.141	-0.7042
Make sure you :) or :D today!	0.895	0.8633
Not bad at all	0.307	0.431

There is a slight difference between the values especially for words including negation and intensifiers. This is because we assume a constant value for the negation and intensifier words, whereas the Python implementation has a more complex algorithm to calculate the sentiment value of the word. Moreover, our negation algorithm, as explained in the algorithm section, works all the way until another negation word is found, whereas the Python implementation is a little more advanced in this regard. Also, our algorithm only considers exclamation marks if the word they follow is a sentiment word, whereas the Python implementation considers them in context of the full sentence. That being said, both the C and the Python implementations manage to agree on whether or not a sentence is positive (positive value) or negative (negative value). Therefore, we can conclude that both implementations, though different in their approach, are able to achieve roughly the same consensus on the sentiment of the sentence.

6 Appendix

Now I will show all the code files used in the program.

6.1 utility.h

The utility.h file takes a lot of the constants used throughout the program, as well as the struct definition and function prototypes.

```
#ifndef UTILITY_H
#define UTILITY_H

// Define constants
#define ARRAY_SIZE 20
#define MAX_STRING_LENGTH 200
#define LINE_LENGTH 100
```

```
// Positive words array
#define POSITIVE_INTENSIFIERS_SIZE 11
static char *positive_intensifiers[] = {
        "absolutely",
        "completely",
        "extremely",
        "really",
        "so",
        "totally",
        "very",
        "particularly",
        "exceptionally",
        "incredibly",
        "remarkably",
};

// Negative words array
#define NEGATIVE_INTENSIFIERS_SIZE 9
static char *negative_intensifiers[] = {
        "barely",
        "hardly",
        "scarcely",
        "somewhat",
        "mildly",
        "slightly",
        "partially",
        "fairly",
        "pretty-much",
};

// Negation words array
#define NEGATIONS_SIZE 13
static char *negation_words[] = {
        "not",
        "isn't",
        "doesn't",
        "wasn't",
        "shouldn't",
        "won't",
        "cannot",
        "can't",
        "nor",
        "neither",
        "without",
```

```
        "lack",
        "missing",
};

// The boost from different amplifiers
#define INTENSIFIER 0.293
#define EXCLAMATION 0.292
#define CAPS 1.5
#define NEGATION -0.5


// Include necessary libraries
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>
#include <math.h>

// Define the structure for the data
typedef struct {
        char word[MAX_STRING_LENGTH];
        float value1;
        float value2;
        int intArray[ARRAY_SIZE];
} WordData;

// Function prototypes
WordData *read_data(char *filename);
float *calculate_sentiment_score(WordData *data, char *sentence);
WordData find_data(WordData *data, char *word);

#endif
```

6.2 vaderSentiment.c

The vaderSentiment.c file contains the implementations of all of the functions used for the purpose of calculating the sentiment of the sentence.

```
#include "utility.h"

WordData* read_data(char *filename) {

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

    // Ensure the file was successfully opened
    if (file == NULL) {
        printf("Error opening file\n");
                exit(1);
    }
```

```
    // Allocate memory for the first block of WordData
    WordData *data = malloc(sizeof(WordData));
    if (data == NULL) {
        printf("Memory allocation failed\n");
        fclose(file);
                exit(1);
    }

    int i = 0;

    // Read data from the file
    while (1) {

                // Allocate memory for the WordData item
                data = realloc(data, (i+1) * sizeof(WordData));

                // Ensure memory was allocated
                if (data == NULL) {
                        printf("Memory allocation failed\n");
                        free(data);
                        fclose(file);
                        exit(1);
                }

        // Read the word and two float values
        fscanf(file, "%s %f %f", &data[i].word, &data[i].value1, &data[i].value2);

                // Break if end of file
                if (feof(file)) {
                        data[i].word[0] = '\0';
                        break;
                }

        // Read the integer array
        char line[2000];
        fgets(line, 2000, file);  // Read the line for integers

        char *token = strtok(line, "[], \n\t\v\f\r");

        for (int j = 0; j < ARRAY_SIZE && token != NULL; j++) {
            data[i].intArray[j] = atoi(token);
            token = strtok(NULL, "[], \n\t\v\f\r");
        }

        // Prepare to read the next WordData item
        i++;

    }

    // Close the file
    fclose(file);

        // Return the data
        return data;
}

// Linear search for a word in the WordData array
WordData find_data(WordData *data, char *word) {

        // Search for the word in the data array
        for (int i = 0; data[i].word[0] != '\0'; i++) {
                if (strcmp(data[i].word, word) == 0) {
                        return data[i];
                }
        }

        // Return a WordData item with a null word if word not found
        WordData nullData;
        nullData.word[0] = '\0';
        return nullData;
}

// Function to calculate the sentiment score of a sentence
float *calculate_sentiment_score(WordData *data, char *sentence) {

        // Scores array
        float scores[MAX_STRING_LENGTH];

        // Count of all words with sentiment
        int sentimentCount = 0;

        // Sum of all sentiment scores
        float sentimentSum = 0.0;

        // Variables to store calculations for pos, neu, neg
        float pos = 0.0;
        float neu = 0.0;
        float neg = 0.0;

        // Stores the current intensifier values
        float intensifier = 0;
```

```
// Stores the current negation value
float negation = 0;

// Array to store the split sentence
char sentence_split[MAX_STRING_LENGTH][MAX_STRING_LENGTH];

// Copy the sentence for strtok to work
char sentence_copy[MAX_STRING_LENGTH];
strcpy(sentence_copy, sentence);

// Since the only punctuation we care about is exclamation marks
// We will remove all other punctuation
char *token = strtok(sentence_copy, "-\n\t\v\f\r,.?");

// Loop through the tokens
for (int index = 0; token != NULL; index++) {

        // Booleans to check for all caps and exclamations
        bool allCaps = true;
        int exclamation = 0;

        // Convert the token to lowercase
        char currWord[MAX_STRING_LENGTH];
        strcpy(currWord, token);

        // Iterate through each character and use tolower()
        for (int i = 0; currWord[i] != '\0'; i++) {

                // Check for all uppercase and ensure exclamation marks are not counted
                if (islower(currWord[i]) && currWord[i] != '!') allCaps = false;

                // Convert to lowercase
                currWord[i] = tolower(currWord[i]);

                // Check for exclamation marks
                if (currWord[i] == '!') {

                        // Increment the exclamation count and null the character
                        exclamation++;
                        currWord[i] = '\0';

                        // Limit the exclamation count to 3
                        if (exclamation > 3) exclamation = 3;

                }
        }

        // Copy the token to the sentence split array
        strcpy(sentence_split[index], currWord);

        // Find the word in the data array
        WordData wordData = find_data(data, currWord);

        // Check if the word was found
        if (wordData.word[0] != '\0') {

                // Increment the sentiment count
                sentimentCount++;

                // Add the score to the scores array
                scores[index] = wordData.value1;

                // If word is capitalized, multiply by factor
                if (allCaps) {
                        scores[index] *= CAPS;
                }

                // Add intensifier and negation values
                if (intensifier != 0) scores[index] += intensifier * scores[index];
                if (negation != 0) scores[index] *= negation;

                // Add exclamation mark if positive, and subtract if negative
                // Only works if the current word is a sentiment word
                if (scores[index] > 0) {
                        scores[index] += exclamation * scores[index] * EXCLAMATION;
                } else {
                        scores[index] -= exclamation * scores[index] * EXCLAMATION;
                }
                sentimentSum += scores[index];
        }

        // Reset intensifier. Negation remains until next negation word
        intensifier = 0;

        // Check if current word is an intensifier
        // Check positive intensifiers
        for (int i = 0; i < POSITIVE_INTENSIFIERS_SIZE; i++) {
                if (strcmp(sentence_split[index], positive_intensifiers[i]) == 0) {
                        intensifier = INTENSIFIER;
                }
        }
```

```
                // Check negative intensifiers
                for (int i = 0; i < NEGATIVE_INTENSIFIERS_SIZE; i++) {
                        if (strcmp(sentence_split[index], negative_intensifiers[i]) == 0) {
                                intensifier = -INTENSIFIER;
                        }
                }

                // If intensifier is capitalized, multiply by factor
                if (allCaps && intensifier != 0) {
                        intensifier *= CAPS;
                }

                // Check if current word is a negation
                for (int i = 0; i < NEGATIONS_SIZE; i++) {
                        if (strcmp(sentence_split[index], negation_words[i]) == 0) {
                                negation = NEGATION;
                        }
                }

                // If negation is capitalized, multiply by factor
                if (allCaps && negation != 0) {
                        negation *= CAPS;
                }

                // Calculate pos, neu, neg
                if (scores[index] > 0) {
                        pos += scores[index];
                } else if (scores[index] == 0) {
                        neu += 1;
                } else {
                        neg += -1 * scores[index];
                }

                // Get the next token
        token = strtok(NULL, " \n\t\v\f\r,.?");
        }

        // Calculate the compound score
        float compound = sentimentSum / sqrt( pow(sentimentSum, 2) + 15 );

        // Get the pos, neu, neg values
        float total = pos + neu + neg;
        pos = pos / total;
        neu = neu / total;
        neg = neg / total;

        // Create and return an array of the scores
        float *scoreArray = malloc(4 * sizeof(float));
        scoreArray[0] = compound;
        scoreArray[1] = pos;
        scoreArray[2] = neu;
        scoreArray[3] = neg;

        // Return the array
        return scoreArray;
}
```

6.3 main.c

The main.c file contains the main function, which is used to run the program on the test cases.

```
#include "utility.h"

void main() {

        // Get the data from the file
    WordData *data = read_data("vader_lexicon.txt");

        // Testcases
        char *sentences[] = {
                "VADER is smart, handsome, and funny.",
                "VADER is smart, handsome, and funny!",
                "VADER is very smart, handsome, and funny.",
```

```
            "VADER is VERY SMART, handsome, and FUNNY.",
            "VADER is VERY SMART, handsome, and FUNNY!!!",
            "VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY
                !!!",
            "VADER is not smart, handsome, nor funny.",
            "At least it isn't a horrible book.",
            "The plot was good, but the characters are uncompelling
                and the dialog is not great.",
            "Make sure you :) or :D today!",
            "Not bad at all",
        };

        // Iterate through the testcases and output the sentiment score
            , positive sentiment, neutral sentiment, and negative
            sentiment
        for (int i = 0; i < sizeof(sentences) / sizeof(sentences[0]); i
            ++) {
            float *score = calculate_sentiment_score(data,
                sentences[i]);
            printf("Sentence:\n%s\n", sentences[i]);
            printf("Sentiment score: %f\n", *score);
            printf("Positive sentiment: %f\n", *(score + 1));
            printf("Neutral sentiment: %f\n", *(score + 2));
            printf("Negative sentiment: %f\n\n", *(score + 3));

            // Free the memory allocated for the score
            free(score);
        }

        // Free the memory allocated for the data
        free(data);
}
```

6.4 Makefile

The Makefile is used to compile the program.

```
vader: main.c utility.h vaderSentiment.c
	gcc -o vader main.c utility.h vaderSentiment.c -lm
```

6.5 test.py

The test.py file is used to test the Python implementation of VADER on the same test cases. This uses the `NLTK` library to calculate the sentiment of the sentence.

```
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

```
from nltk import tokenize

nltk.download('vader_lexicon')

sentences = [
    "VADER is smart, handsome, and funny.",
    "VADER is smart, handsome, and funny!",
    "VADER is very smart, handsome, and funny.",
    "VADER is VERY SMART, handsome, and FUNNY.",
    "VADER is VERY SMART, handsome, and FUNNY!!!",
    "VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY!!!",
    "VADER is not smart, handsome, nor funny.",
    "At least it isn't a horrible book.",
    "The plot was good, but the characters are uncompelling and the dialog is not g
    "Make sure you :) or :D today!",
    "Not bad at all",
]

# Output the vader sentiment analysis of the above sentences
sid = SentimentIntensityAnalyzer()
for sentence in sentences:
    print(sentence)
    ss = sid.polarity_scores(sentence)
    for k in sorted(ss):
        print('{0}: {1}, '.format(k, ss[k]), end='')
    print("\n")
```