Assignment 3 Report: Implementing VADER Sentiment Analysis in C

MECHTRON 2MP3: Programming for Mechatronics

Furqaan Khurram Qamar
400514719

Algorithms Implemented:

These are the algorithms implemented in this assignment:

1. Reading Data from File (read_data):

This function loads the sentiment words and their scores from a file into an array of structs. Think of it like building a list of words where each word has its own "happiness" or "sadness" score, which we use later for analyzing sentences.

Algorithm:

- Open the file (vader_lexicon.txt) with fopen.
- Start with an empty list of WordData (using malloc to get some initial memory).
- Loop through each line of the file:
    - Read a whole line using getline (safer than fscanf because it avoids overflow).
    - Split the line into parts to grab the word and its scores using sscanf.
    - Store this info in the array and use realloc to expand the array for the next word.
- Finally, close the file and return the array of data.

Time Complexity:

- It's **O(n)** because we go through every line in the file once

2. Calculating Sentiment (calculate_sentiment_score):

This function takes a sentence and figures out if it's positive, negative, or neutral based on the words. It uses the scores from the lexicon data we loaded earlier.

Algorithm:

- Split the sentence into individual words (using strtok).
- For each word:

- Convert it to lowercase so "Happy" and "happy" are treated the same.
- Check if it's an **intensifier** (like "very" or "extremely") and adjust the score accordingly.
- Check if it's a **negation** word (like "not"), and if so, reverse the sentiment (positive becomes negative).
- Search the word in our `WordData` list to get its score.
- Add the score to the running total (sentimentSum) if it's found.
- If there are exclamation marks:
  - The sentimentSum score is adjusted based on how many there are. For each !, the score gets boosted slightly by 0.292.
  - Specifically, if the score is positive, we **add** a small boost, and if it's negative, we **subtract** a small boost.
  - This makes the sentence more intense based on the number of !.
- There is also a bonus feature added, much like the VADER program in Python that does the following:
  - **Sentiment Score Calculation**: Each word's sentiment score is added to `pos_sum` if positive, `neg_sum` if negative, or `neutral` if not found in the data, allowing you to capture the proportion of positive, negative, and neutral words in the sentence.
  - **Percentage Calculation**: The total counts (`pos_sum`, `neg_sum`, and `neutral`) are divided by their sum to get the relative percentages of positive, negative, and neutral sentiment for the sentence.

- Finally, calculate a **compound score** using:

$$compound = \frac{sentimentSum}{\sqrt{sentimentSum^2 + 15}}$$

- This formula helps scale the score between -1 (very negative) and 1 (very positive).

- Also prints the Positive, Negative, Neutral sentiment as well as the compound sentiment of the sentence.

- Return this compound score.

Time Complexity:

- The worst-case scenario is **O(n*m)**, where `n` is the number of words in the sentence, and `m` is the number of words in our lexicon. This is because for each word in the sentence, we might have to search through all words in the lexicon.

3. Finding Data in the Lexicon (`find_data`):

This function searches our array of `WordData` to find a matching word and get its score.

Algorithm:

- Loop through the `WordData` array and compare each word with the one we're searching for using `strcmp`.
- If a match is found, return the `WordData` struct with the word's scores.
- If no match is found, return an empty or default struct.

Time Complexity:

- It's **O(m)** in the worst case, where `m` is the number of words in our `WordData` array because we might need to check each word once.

Summary

The program reads in a list of words with their sentiment scores, analyzes sentences by checking each word, and then gives a score based on how positive or negative the words are. The main work is in reading the data and then looping through words in sentences to figure out their sentiment scores

Compiling and Running:

The C code provided below should compile and run on any operating system that has a C compiler, such as GCC or Clang. Firstly on an IDE or text editor, open up **main.c**, making sure that the **Makefile**, **utility.h**, **main.c** and **vaderSentiment.c** are in the same folder.

To compile and run the code in the terminal using GCC we can type the following into the terminal:

```
make
./sentiment_analyzer
```

The code will now run and provide the compound score of the sentiment analysis of the input sentences.

Case Study with Vader developed in C:

Sentence	Compound	
	Model in C	**Python Library**
VADER is smart, handsome, and funny.	0.831632	0.8316
VADER is smart, handsome, and funny!	0.843896	0.8439
VADER is very smart, handsome, and funny.	0.851826	0.8545
VADER is VERY SMART, handsome, and FUNNY.	0.913924	0.9227
VADER is VERY SMART, handsome, and FUNNY!!!	0.927330	0.9342
VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY!!!	0.927330	0.9469
VADER is not smart, handsome, nor funny.	-0.599373	-0.7424
At least it isn't a	-0.542326	-0.5423

horrible book.		
The plot was good, but the characters are uncompelling and the dialog is not great.	-0.140599	-0.7042
Make sure you :) or :D today!	0.835634	0.8633
Not bad at all	0.307148	0.431

The main difference between my C implementation and the Python vaderSentiment library is how we handle some of the finer details in text processing. While I made sure to account for punctuation like exclamation marks (!) to boost sentiment intensity, the Python version goes a bit further. It deals with things like repeated letters (e.g., "sooo happy"), slang (“FRIGGIN”), and even emojis, which I didn't include in my version. Python's VADER also looks at short phrases (like "not good") to adjust scores based on context. Because of these extra features, the Python implementation can capture subtle emotional nuances that mine might miss, leading to slight differences in the final sentiment scores.

Appendix:

References:

OpenAI. (2024). *ChatGPT* (Mar 14 version) [Large language model]. https://chat.openai.com/chat

Source Code:

main.c

```
#include "utility.h"

int main() {
   WordData *data = read_data("vader_lexicon.txt");

   // Define an array of sentences
   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"
   };


   // Calculate the number of sentences
   int num_sentences = sizeof(sentences) / sizeof(sentences[0]);

   // Iterate over each sentence and calculate the sentiment score
   for (int i = 0; i < num_sentences; i++) {
       printf("Sentence: %s\n", sentences[i]);
       float *scores = calculate_sentiment_score(data, sentences[i]);
```

```
        printf("Sentiment Score: (Positive: %f, Negative: %f, Neutral: %f,
Compound: %f)\n\n", scores[0], scores[1], scores[2], scores[3]);
        free(scores);
    }

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

utility.h

```
#ifndef UTILITY_H
#define UTILITY_H

// Define general constants
#define ARRAY_SIZE 20               // Array size for intArray in WordData
struct
#define MAX_STRING_LENGTH 200       // Maximum length for strings
#define LINE_LENGTH 100             // Maximum length of a line in the file

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

// Positive intensifiers that amplify positive sentiment
#define POSITIVE_INTENSIFIERS_SIZE 11
static char *positive_intensifiers[] = {
    "absolutely",
    "completely",
    "extremely",
    "really",
    "so",
    "totally",
    "very",
    "particularly",
    "exceptionally",
```

```
    "incredibly",
    "remarkably",
};

// Negative intensifiers that slightly reduce positive or amplify negative
sentiment
#define NEGATIVE_INTENSIFIERS_SIZE 9
static char *negative_intensifiers[] = {
    "barely",
    "hardly",
    "scarcely",
    "somewhat",
    "mildly",
    "slightly",
    "partially",
    "fairly",
    "pretty much",
};

// Words indicating negation, which invert the sentiment of the following
word
#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",
};

// Constants for sentiment adjustment
```

```c
#define INTENSIFIER 0.293          // Multiplier for intensifiers
(positive or negative)
#define EXCLAMATION 0.292          // Boost from exclamation marks
#define CAPS 1.5                   // Boost for words in all caps
#define NEGATION -0.5              // Factor to invert sentiment on
negated words

// Structure to hold word data, including sentiment scores and an integer
array
typedef struct {
   char word[MAX_STRING_LENGTH];  // Word string
   float value1;                  // Primary sentiment score
   float value2;                  // Secondary sentiment score
   int intArray[ARRAY_SIZE];      // Integer array (customizable for
additional data)
} WordData;

// Function prototypes
WordData *read_data(char *filename);              // Reads WordData from
a file
float *calculate_sentiment_score(WordData *data, char *sentence);  //
Calculates sentiment score for a sentence
WordData find_data(WordData *data, char *word);    // Searches for a word
in the WordData array
int is_all_caps(const char* word); // Returns true if word is all caps

#endif
```

vaderSentiment.c

```c
#include "utility.h"
#include <errno.h>



int containsPunctuationExceptExclamation(const char *str) {
   while (*str != '\0') {
       if (ispunct(*str) && *str != '!') {
           return 1;  // Found punctuation other than '!', return true
```

```
        }
        str++;  // Move to the next character
    }
    return 0;  // No punctuation other than '!' found
}

int is_all_caps(const char* word) {
    for (int j = 0; word[j] != '\0'; j++) {
        if (!isupper(word[j])) {
            return 0;  // Return false if any character is not uppercase
        }
    }
    return 1;  // Return true if all characters are uppercase
}


// Reads data from a file and stores it in an array of WordData structs
WordData* read_data(char *filename) {
    FILE *file = fopen(filename, "r");

    // Check if file opened successfully
    if (!file) {
        perror("Error opening file");
        return NULL;
    }

    // Allocate initial memory for WordData
    WordData *data = malloc(sizeof(WordData));
    if (!data) {
        perror("Memory allocation failed");
        fclose(file);
        return NULL;
    }

    size_t line_size = LINE_LENGTH;
    char *line = malloc(line_size);
    int i = 0;

    // Read each line from the file and store it in the WordData array
    while (getline(&line, &line_size, file) != -1) {
```

```
        data = realloc(data, (i + 1) * sizeof(WordData));
        sscanf(line, "%s %f %f", data[i].word, &data[i].value1,
&data[i].value2);
        i++;
    }

    free(line);
    fclose(file);
    return data;
}

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

    // Loop through data to find the word
    for (int i = 0; data[i].word[0] != '\0'; i++) {
        if (strcmp(data[i].word, word) == 0) {
            return data[i];
        }
    }

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

// Calculates the sentiment score of a sentence based on word data
#include <math.h>

// Modify function to return an array of sentiment scores
float* calculate_sentiment_score(WordData *data, char *sentence) {
    float scores[MAX_STRING_LENGTH] = { 0.0 };
    int index = 0;
    int sentimentCount = 0;
    float sentimentSum = 0.0;
    float pos_sum = 0.0;
    float neg_sum = 0.0;
    float neutral = 0.0;
    char sentence_split[MAX_STRING_LENGTH][MAX_STRING_LENGTH];
```

```
    char sentence_split_original[MAX_STRING_LENGTH][MAX_STRING_LENGTH];
    char sentence_copy[MAX_STRING_LENGTH];
    int total_exclamations = 0; // Track total exclamations

    strcpy(sentence_copy, sentence);
    char *token = strtok(sentence_copy, " \n\t\v\f\r,.?");

    int isNegated = 0; // Initialize negation flag
    for (; token != NULL; index++) {

        bool allCaps = true;
        int exclamation = 0;
        strcpy(sentence_split_original[index], token);
        char lowerToken[MAX_STRING_LENGTH];
        strcpy(lowerToken, token);

        for (int i = 0; lowerToken[i] != '\0'; i++) {
            if (islower(lowerToken[i])) allCaps = false;
            lowerToken[i] = tolower(lowerToken[i]);

            if (lowerToken[i] == '!') {
                exclamation++;
                lowerToken[i] = '\0';
            }
        }

        total_exclamations += exclamation;
        strcpy(sentence_split[index], lowerToken);
        WordData wordData = find_data(data, lowerToken);

        // Check for negation words and set negation flag
        for (int i = 0; i < NEGATIONS_SIZE; i++) {
            if (strcmp(lowerToken, negation_words[i]) == 0) {
                isNegated = 1;
                break;
            }
        }

        // Check for neutral words; only reset negation on actual sentiment
words
```

```
        if (strcmp(lowerToken, "and") == 0 || strcmp(lowerToken, ",") == 0
|| strcmp(lowerToken, "a") == 0) {
            token = strtok(NULL, " \n\t\v\f\r,.?");
            continue;
        }

        // If relevant word found, apply sentiment score
        if (wordData.word[0] != '\0') {
            sentimentCount++;
            scores[index] = wordData.value1;

            // Apply CAPS multiplier
            if (allCaps &&
!containsPunctuationExceptExclamation(sentence_split[index])) {
                scores[index] *= CAPS;
            }

            // Apply negation if flagged
            if (isNegated) {
                scores[index] *= NEGATION;
                // Do not reset `isNegated` here; allow it to persist to
the next word in this phrase
            }

            if (index > 0) {
                float intensifier_multiplier = INTENSIFIER;
                for (int i = 0; i < POSITIVE_INTENSIFIERS_SIZE; i++) {
                    if (strcmp(sentence_split[index - 1],
positive_intensifiers[i]) == 0) {
                        if (is_all_caps(sentence_split_original[index -
1])) {
                            intensifier_multiplier *= CAPS;
                        }
                        scores[index] += scores[index] *
intensifier_multiplier;
                    }
                }

                for (int i = 0; i < NEGATIVE_INTENSIFIERS_SIZE; i++) {
```

```
                if (strcmp(sentence_split[index - 1],
negative_intensifiers[i]) == 0) {
                    if (is_all_caps(sentence_split_original[index -
1])) {
                        intensifier_multiplier *= CAPS;
                    }
                    scores[index] -= scores[index] *
intensifier_multiplier;
                }
            }
        }


        // Add score to positive, negative, or neutral sums
        if (scores[index] > 0) {
            pos_sum += scores[index];
        } else if (scores[index] < 0) {
            neg_sum += fabs(scores[index]);
        } else {
            neutral += 1;
        }

        sentimentSum += scores[index];
    } else {
        neutral += 1;
    }
    token = strtok(NULL, " \n\t\v\f\r,.?");
}

// Adjust final sentimentSum for exclamations
if (sentimentSum > 0) {
    sentimentSum += total_exclamations * EXCLAMATION;
} else if (sentimentSum < 0) {
    sentimentSum -= total_exclamations * EXCLAMATION;
}

float pos_percent = pos_sum / (pos_sum + neg_sum + neutral);
float neg_percent = neg_sum / (pos_sum + neg_sum + neutral);
float neu_percent = neutral / (pos_sum + neg_sum + neutral);
float compound = sentimentSum / sqrt(pow(sentimentSum, 2) + 15);
```

```
    // Allocate array for returning the scores
    float *result = malloc(4 * sizeof(float));
    if (!result) {
        perror("Memory allocation failed");
        return NULL;
    }

    result[0] = pos_percent;
    result[1] = neg_percent;
    result[2] = neu_percent;
    result[3] = compound;

    return result;
}
```

Makefile

```
# Compiler
CC = gcc

# Compiler flags
CFLAGS = -g -Wall -Wno-unused-variable

# Target executable name
TARGET = sentiment_analyzer

# Source files
SRCS = main.c vaderSentiment.c

# Object files
OBJS = $(SRCS:.c=.o)

# Default rule to build the target
all: $(TARGET)

# Rule to build the target executable
$(TARGET): $(OBJS)
    $(CC) $(OBJS) -o $(TARGET) $(CFLAGS) -lm
```

```
# Rule to compile each .c file to .o
%.o: %.c
   $(CC) $(CFLAGS) -c $< -o $@

# Clean rule to remove compiled files
clean:
   rm -f $(TARGET) $(OBJS)

# Valgrind rule to check for memory leaks
valgrind: $(TARGET)
   valgrind --leak-check=full --track-origins=yes ./$(TARGET)
```

vaderSentiment.py

```
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer

# Initialize the analyzer
analyzer = SentimentIntensityAnalyzer()

# Example texts
texts = [
   "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"
]

# Get sentiment scores
compound_scores = [analyzer.polarity_scores(text)['compound'] for text in
texts]

# Print the compound scores
print(f"Compound Sentiment Scores: {compound_scores}")
```