Assignment 3

Implementing VADER Sentiment Analysis in C

Keerath Singh

MECHTRON 2MP3

400506284

Nov, 14th, 2024

Introduction:

The VADER (Valence Aware Dictionary and sentiment Reasoner) sentiment analysis implementation is designed to measure the sentiment of sentences based on lexical features and contextual cues. The analysis evaluates the emotional tone of a text and produces a compound score that indicates overall sentiment intensity, as well as separate scores for positive, negative, and neutral sentiment.

This report will describe the approach used in the implementation, highlighting the code structure and explaining each component in detail.

Overall Problem Statement

The project aims to create a sentiment analysis tool that can automatically evaluate sentences and return quantitative measures of sentiment. Specifically, the system needs to calculate the following:

1. **Positive Sentiment Score**: Quantifies the overall positivity in the text.
2. **Negative Sentiment Score**: Quantifies the overall negativity in the text.
3. **Neutral Sentiment Score**: Measures the proportion of text with no discernible positive or negative sentiment.
4. **Compound Score**: A single, overall sentiment score ranging from -1 (most negative) to +1 (most positive).

Code Structure Overview

The implementation is split into three main files:

1. **`main.c`**: Contains the driver code to run the sentiment analysis on a set of predefined test sentences.
2. **`vaderSentiment.c`**: Contains the core functionality for reading the lexicon and calculating the sentiment scores of individual sentences.
3. **`utility.h`**: Holds common definitions, data structures, and constants used across the program.

Algorithms used in `main.c`

The `main.c` file provides the entry point for our sentiment analysis program. Here is a breakdown of its components:

- **Lexicon Reading Algorithm**:
  The function `read_lexicon_file("vader_lexicon.txt", &word_count)` reads the sentiment lexicon from a file, which contains words and their associated sentiment scores. The lexicon serves as the foundation for analyzing the sentiment of the input sentences.

- **Algorithm for Running Sentiment Analysis**:
  The function `calculate_sentiment_score(test_sentences[i], lexicon, word_count)` is called for each sentence. This function analyzes each word within the sentence and modifies its sentiment based on contextual cues such as negations, intensifiers, capitalization, and punctuation. The final sentiment for the sentence is then computed.

- **Memory Management**:
  Dynamic memory is allocated for storing the lexicon (`lexicon`) and is freed after the analysis to avoid memory leaks.

Algorithms in `vaderSentiment.c`

The `vaderSentiment.c` file contains the core functions for sentiment analysis:

- **Word Lookup Algorithm**:
  The `find_data()` function performs a linear search on the lexicon array to find the sentiment value of a word. This is used during sentiment analysis to determine if a word in the sentence matches an entry in the lexicon.

  Process:

  - It takes the lexicon array (`WordData *data`) and the word to search (`char *word`) as arguments.
  - For each entry in the lexicon, it compares the word in the lexicon with the target word using `strcmp()`.

    - If a match is found, it returns the `WordData` structure containing the sentiment values for the word.
    - If no match is found, it returns a null `WordData` structure.

- **Sentiment Calculation Algorithm**: The `calculate_sentiment_score()` function takes a sentence and the lexicon as input and calculates four types of sentiment scores:

- **Intensifiers Handling**
    - Words like "very" or "extremely" amplify the sentiment of the following word.
    - If a word is an intensifier, its value is applied to adjust the next word's sentiment score using a multiplier (`INTENSIFIER = 0.293`).

- **Negations Handling**
    - Words like "not" or "never" invert the sentiment of the following word.
    - This is implemented by multiplying the sentiment score by a negative factor (`NEGATION = -0.5`).

- **Capitalization Handling**
    - Words in **ALL CAPS** have increased intensity.
    - The sentiment score is amplified by a factor (`CAPS = 1.5`) for such words.

- **Punctuation Handling**
    - **Exclamation marks** add emphasis.
    - Sentiment value is adjusted by adding/subtracting a factor (`EXCLAMATION = 0.292`) based on the number of exclamations (limited to a maximum of 3).

- After adjusting the sentiment scores based on these contextual cues, the function calculates the overall sentiment of the sentence. The **compound score** is computed using a normalization function to ensure that the value lies between -1 and +1. Formula used:

$$compound = \frac{total\ score}{\sqrt{total\ score^2 + \alpha}}$$

Alpha default value set to 15.

- **Sifting Sentiment Scores Algorithm**: The `sift_sentiment_scores()` function divides the word sentiment values into positive, negative, and neutral components. These scores are then normalized based on the total sentiment value.

Explanation of `utility.h`

The `utility.h` file provides essential definitions, data structures, and function prototypes that form the backbone of the VADER sentiment analysis implementation:

1. Constants

- **Modifiers and Amplifiers:**
  - `Explained in section above (code is defined in utility)`
    SEE APENDIX FOR FULL CODE DETAIL INCLUDING WORDS USED

2. Data Structures

- **`WordData` Struct:**
  - Represents each word in the lexicon.
  - Fields include:
    - `word (char[MAX_STRING_LENGTH])`: Stores the actual word.
    - `value1 (float)`: Represents the sentiment score for the word.
    - `value2 (float)`: Used for additional sentiment information.
- **`SentimentResult` Struct:**
  - Holds the calculated sentiment analysis result for a sentence.
  - Fields include:

        - `pos (float)`: Positive sentiment score.
        - `neg (float)`: Negative sentiment score.
        - `neu (float)`: Neutral sentiment score.
        - `compound (float)`: Normalized composite score representing the overall sentiment.

3. Function Prototypes

- **Lexicon Management:**
    - `read_lexicon_file()`: Reads words from the lexicon file and stores them in an array of `WordData` structs.
- **Sentiment Analysis:**
    - `calculate_sentiment_score()`: Analyzes the sentiment of a given sentence, calculating the `pos`, `neg`, `neu`, and `compound` scores.
- **Data Lookup:**
    - `find_data()`: Searches for a word in the lexicon and returns its sentiment values.
- **Utility Functions:**
    - `is_all_caps()`: Checks if a given word is in all uppercase, contributing to the intensity of the sentiment.
    - `sift_sentiment_scores()`: Splits the calculated sentiment scores into positive, negative, and neutral components.

Libraries Used

1. #include <stdio.h>

Provides functions for input and output, such as printf() and fopen(), which are used for reading files and printing results.

2. #include <string.h>

Contains functions for string manipulation, like strcpy(), strcmp(), and strtok(), which are used for copying, comparing, and tokenizing strings.

3. #include <stdlib.h>

Provides functions for memory management (malloc(), realloc(), free()), as well as

conversion functions like atoi(). It is crucial for dynamic allocation of memory.

4. #include <stdbool.h>

Adds support for boolean data types (true and false), making the code easier to read and use conditional statements effectively.

5. #include <ctype.h>

Contains functions to classify and manipulate individual characters, such as isupper(), tolower(), and islower(). These are used to process words for lexicon lookup and identify uppercase words.

6. #include <errno.h>

Provides macros to report error numbers (errno). Useful for debugging errors, especially when handling file I/O operations like opening a lexicon file.

7. #include <math.h>

Contains mathematical functions like sqrt() and fabs(), which are used to normalize and compute sentiment scores.

ChatGPT Utilization

ChatGPT was used in canvas mode, which allows you to write and debug code directly with assistance from ChatGPT

Run and Compile

To run the program use the command in a unix-based terminal:
gcc main.c vaderSentiment.c -o vaderSentiment

Followed by:
./vaderSentiment

Alternativily, we can use a make file:
make

Compound Score Analysis

Sentence	Compound Score (C)	Compound Score (Python)
VADER is smart, handsome, and funny	0.831632	0.8316
VADER is smart, handsome, and funny!	0.853909	0.8439
VADER is very smart, handsome, and funny.	0.851826	0.8545
VADER is VERY SMART, handsome, and FUNNY.	0.907112	0.9227
VADER is VERY SMART, handsome, and FUNNY!!!	0.941736	0.9342
VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY!!!	0.941736	0.9469
VADER is not smart, handsome, nor funny.	-0.599373	-0.7424
At least it isn't a horrible book.	-0.542326	-0.5423
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!	0.894520	0.8633
Not bad at all	0.307148	0.431

Time and Space Complexity Analysis

The time complexity of the `calculate_sentiment_score` function is **O(n)**, where **n** is the number of words in the input sentence, because the function processes each word sequentially in a single pass.the value range, making it more consistent for varying data sets. The space complexity of the `calculate_sentiment_score` function is **O(n)**, where **n** is the number of words in the input sentence, because it allocates arrays and uses variables that scale linearly with the number of words to store intermediate sentiment scores and other data.

References

[1] C.J. Hutto and E.E. Gilbert, "VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text," *Eighth International Conference on Weblogs and Social Media (ICWSM-14)*, Ann Arbor, MI, June 2014. Available: https://github.com/cjhutto/vaderSentiment.

[2] OpenAI, *ChatGPT (Version GPT-4o & Io Preview)*, OpenAI, San Francisco, CA, Oct. 2024. Available: https://openai.com/chatgpt

Appendix:

Main.c

```c
#include <stdio.h>     // Standard Input/Output library for printing output to the console

#include <stdlib.h>    // Standard Library for memory management (malloc, free)

#include <string.h>    // String handling functions (strcpy, strcmp, etc.)

#include <stddef.h>    // Added for size_t, needed for array indexing, a good practice to use size_t to avoid overflow

#include "utility.h"   // Our custom header file containing definitions, structures, and function prototypes


int main() {

   int word_count = 0;  // Variable to keep track of the number of words in our lexicon


   // Read the lexicon file

   // Here, we call the `read_lexicon_file()` function to load the sentiment lexicon data from the file.

   // If there's an error (e.g., the file is missing), it will print an error message and exit the program.

   WordData *lexicon = read_lexicon_file("vader_lexicon.txt", &word_count);

   if (lexicon == NULL) {

       // If the file couldn't be read, print error and exit.

       printf("Error: Unable to rread lexicon file.\n");
```

```
        return 1;  // Returning a non-zero value here indicates abnormal
termination due to error
    }


    // Test Sentences
    const char *test_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"
    };


    // Run sentiment analysis on each test sentence
    // The loop runs through each test sentence and performs sentiment
analysis.
```

```
   // We use `sizeof(test_sentences) / sizeof(test_sentences[0])` to get
the total number of sentences,
   // which ensures compatibility if the array size changes.
   for (size_t i = 0; i < sizeof(test_sentences) /
sizeof(test_sentences[0]); i++) {
       // Calculate the sentiment score for each sentence using our
sentiment analysis function
       // `calculate_sentiment_score()` returns a `SentimentResult`
structure containing `neg`, `neu`, `pos`, and `compound` scores.
       SentimentResult sentiment_score =
calculate_sentiment_score(test_sentences[i], lexicon, word_count);

       // Printing the results of the sentiment analysis
       // We use `%f` here to format the floating-point scores. Since
VADER produces four key scores (positive, negative, neutral, and
compound),
       // we display all of them for each sentence.
       printf("Sentence: \"%s\"\n", test_sentences[i]);
       printf("{'neg': %f, 'neu': %f, 'pos': %f, 'compound': %f}\n\n",
              sentiment_score.neg, sentiment_score.neu,
sentiment_score.pos, sentiment_score.compound);
   }

   // Free the allocated memory
   // Since we allocated memory for `lexicon` dynamically in
```

```
`read_lexicon_file`, we need to free it after we're done.
    free(lexicon);

    return 0; // Returning 0 indicates successful completion of the program
}
```

utility.h

```
#ifndef UTILITY_H
#define UTILITY_H

// Define general constants
#define ARRAY_SIZE 20              // Array size for intArray in WordData struct; used to hold additional metadata for each word
#define MAX_STRING_LENGTH 200      // Maximum length for strings to accommodate long words or sentences
#define LINE_LENGTH 100            // Maximum length of a line in the file; this helps when reading files

// Include necessary libraries
#include <stdio.h>    // For file operations, input/output
#include <string.h>   // For string manipulation (strcpy, strcmp, etc.)
#include <stdlib.h>   // For dynamic memory allocation (malloc, free, etc.)
```

```c
#include <ctype.h>    // For character checking and manipulation (tolower, isupper, etc.)
#include <stdbool.h>  // For boolean data type
#include <math.h>     // For mathematical calculations used in sentiment analysis


// Positive intensifiers that amplify positive sentiment
// The size is defined as POSITIVE_INTENSIFIERS_SIZE for easy reference
#define POSITIVE_INTENSIFIERS_SIZE 11
static char *positive_intensifiers[] = {
  "absolutely",
  "completely",
  "extremely",
  "really",
  "so",
  "totally",
  "very",
  "particularly",
  "exceptionally",
  "incredibly",
  "remarkably"
};
```

```
// Positive intensifiers are words that amplify positive sentiment in a
sentence. For example, "really happy" is more positive than just "happy."


// 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"
};
// Negative intensifiers weaken a positive sentiment or amplify a negative
one. For instance, "barely good" is much less positive than just "good."


// 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"
};
// Negation words are used to flip the sentiment of a following word. For example, "not happy" means the opposite of "happy."


// Constants for sentiment adjustment
#define INTENSIFIER 0.293          // Multiplier for intensifiers (positive or negative) — this value is empirically derived to adjust sentiment scores
#define EXCLAMATION 0.292          // Boost from exclamation marks to
```

```
indicate emphasis, adding to the overall sentiment score

#define CAPS 1.5                    // Boost for words in all caps,
indicating a higher intensity (e.g., "HAPPY" vs "happy")

#define NEGATION -0.5               // Factor to invert sentiment on
negated words; a negative sentiment value is multiplied by this to invert
its meaning



// Structure to hold word data, including sentiment scores and an integer
array

typedef struct {

  char word[MAX_STRING_LENGTH];  // The word as a string

  float value1;                  // Primary sentiment score (usually
positive or negative)

  float value2;                  // Secondary sentiment score (could be
for other attributes, e.g., emotional intensity)

  int intArray[ARRAY_SIZE];      // Additional data related to the word;
customizable for different use cases

} WordData;

// The WordData struct holds the core data for each word in the sentiment
lexicon. This includes two float scores for sentiment and an integer array
to hold extra info if needed.



// Structure to hold the sentiment analysis results

typedef struct {

   float pos;      // Positive sentiment score
```

```
    float neg;      // Negative sentiment score
    float neu;      // Neutral sentiment score
    float compound; // Compound score, representing the overall sentiment
of the sentence
} SentimentResult;

// SentimentResult is used to store the result of the sentiment analysis
for a given sentence. It includes individual positive, negative, and
neutral scores, as well as a compound score.


// Function prototypes
WordData *read_lexicon_file(const char *filename, int *word_count); //
Reads the lexicon file and returns an array of WordData with the count of
words
SentimentResult calculate_sentiment_score(const char *sentence, WordData
*lexicon, int word_count); // Analyzes the sentiment of a given sentence
WordData find_data(WordData *data, char *word);          // Searches for a
specific word in the WordData array, returning the WordData struct for
that word
int is_all_caps(const char* word);                       // Checks if the
word is in all uppercase letters and returns true if it is


#endif
```

vaderSentiment.c

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


// Reads data from a file and stores it in an array of WordData structs
WordData* read_lexicon_file(const char *filename, int *word_count) {
   FILE *file = fopen(filename, "r"); // Open the file in read mode


   // Error handling: If the file cannot be opened, print an error message and return NULL
   if (!file) {
       perror("Error opening file");
       return NULL;
   }


   int capacity = 100;  // Initial capacity for dynamic array
   WordData *words = malloc(capacity * sizeof(WordData)); // Allocate memory for WordData structs
   if (!words) {
       perror("Memory allocation failed");
       fclose(file);
       return NULL;
```

```
    }

    *word_count = 0; // Initialize word count to 0



    char line[256];

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

        // If the word count exceeds capacity, we need to expand the memory
allocated for words

        if (*word_count >= capacity) {

            capacity *= 2; // Double the capacity

            WordData *temp = realloc(words, capacity * sizeof(WordData));
// Reallocate memory with new capacity

            if (!temp) {

                perror("Memory reallocation failed");

                free(words); // Free the original memory if reallocation
fails

                fclose(file);

                return NULL;

            }

            words = temp; // Point to the newly allocated memory

        }
```

```
        //USED CHATGPT TO GET THE LOGIC RIGHT FOR THIS LINE
        // Extract the word and its sentiment scores from the line
        sscanf(line, "%s %f %f", words[*word_count].word,
&words[*word_count].value1, &words[*word_count].value2);
        (*word_count)++; // Increment word count
    }

    fclose(file); // Close the file after reading all lines
    return words; // Return the dynamically allocated array of WordData
structs
}

// Searches for a specific word in the WordData array
WordData find_data(WordData *data, char *word) {
    for (int i = 0; data[i].word[0] != '\0'; i++) {
        if (strcmp(data[i].word, word) == 0) {
            return data[i]; // Return the WordData struct if the word
matches
        }
    }

    // If no match is found, return a WordData with an empty word
    WordData nullData;
```

```
    nullData.word[0] = '\0'; // Set the first character of word to null to
indicate "not found"
    return nullData;
}


// Helper function to sift sentiment scores into positive, negative, and
neutral components
void sift_sentiment_scores(float sentiments[], int count, float *pos_sum,
float *neg_sum, int *neu_count) {
    *pos_sum = 0.0;  // Initialize positive score sum to zero
    *neg_sum = 0.0;  // Initialize negative score sum to zero
    *neu_count = 0;  // Initialize neutral count to zero


    // Loop through the array of sentiment scores
    for (int i = 0; i < count; i++) {
        if (sentiments[i] > 0) {
            *pos_sum += sentiments[i];  // Accumulate positive scores
        } else if (sentiments[i] < 0) {
            *neg_sum += sentiments[i];  // Accumulate negative scores
        } else {
            (*neu_count)++;  // Increment neutral count for zero scores
        }
    }
```

```
}


//USED CHATGPT TO GET THE LOGIC RIGHT FOR THIS FUNCTION, ALONG WITH PYTHON LIBARIES GIVEN


//USED CHATGPT TO GET TOKEN LOGIC TECHNIQUE, ALONG WITH YOUTUBE TUTORIALS


// Calculates the sentiment score of a sentence and returns a SentimentResult struct
SentimentResult calculate_sentiment_score(const char *sentence, WordData *lexicon, int word_count) {
    float scores[MAX_STRING_LENGTH] = { 0.0 }; // Array to store individual word sentiment scores
    int index = 0; // Index for storing sentiment scores
    float sentimentSum = 0.0; // Cumulative sum of all sentiment scores
    float pos_sum = 0.0, neg_sum = 0.0; // Sums for positive and negative scores
    int neu_count = 0; // Count of neutral scores


    // Copy the sentence into a new array so we can modify it without affecting the original
    char sentence_copy[MAX_STRING_LENGTH];
    strcpy(sentence_copy, sentence);
```

```
   // Tokenize the sentence into individual words
   char *token = strtok(sentence_copy, " \n\t\v\f\r,."); // Delimiters
include spaces, tabs, punctuation


   // Flags for intensifier and negation handling
   bool previous_word_is_intensifier = false; // Tracks if the previous
word was an intensifier
   bool negation_active = false; // Tracks if negation is currently active


   // Loop through each token (word) in the sentence
   while (token != NULL) {
       bool allCaps = true; // Flag to check if the word is in ALLCAPS
       int exclamation = 0; // Counter for the number of exclamation marks


       // Convert token to lowercase for consistent lexicon lookup
       char lowerToken[MAX_STRING_LENGTH];
       strcpy(lowerToken, token);


       for (int i = 0; lowerToken[i] != '\0'; i++) {
           // If the character is lowercase, then the word is not all caps
           if (islower(lowerToken[i])) {
               allCaps = false;
```

```
            }

            // Convert each character to lowercase

            lowerToken[i] = tolower(lowerToken[i]);



            // Count and remove exclamation marks, limiting to max of 3

            if (lowerToken[i] == '!') {

                exclamation++;

                lowerToken[i] = '\0'; // Remove exclamation marks from
token

                if (exclamation > 3) exclamation = 3; // Cap the number of
exclamation marks to 3

            }

        }



        // Find the sentiment value of the word in the lexicon

        WordData wordData = find_data(lexicon, lowerToken);



        // Check if the current token is an intensifier or negation word

        bool is_intensifier = false;

        bool is_negation = false;



        // Check against positive intensifiers
```

```
        for (int i = 0; i < POSITIVE_INTENSIFIERS_SIZE; i++) {
            if (strcmp(lowerToken, positive_intensifiers[i]) == 0) {
                is_intensifier = true;
                break;
            }
        }

        // Check against negation words
        for (int i = 0; i < NEGATIONS_SIZE; i++) {
            if (strcmp(lowerToken, negation_words[i]) == 0) {
                is_negation = true;
                break;
            }
        }


        //USED CHATGPT FOR FLAG METHOD, MY METHOD DID NOT CHECK FULL
SENTENCE AND WOULD GIVEN INCORRECT RESULTS
        // Handle intensifier flag
        if (is_intensifier) {
            previous_word_is_intensifier = true; // Set flag for next word
        } else if (is_negation) {
```

```
            negation_active = true; // Activate negation for following
words

        } else {

            // Process the sentiment value if the word is in the lexicon

            float sentimentValue = 0.0;



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

                sentimentValue = wordData.value1; // Get the primary
sentiment value



                // Amplify sentiment value if the word is in ALLCAPS

                if (allCaps) {

                    sentimentValue *= CAPS; // CAPS amplification factor is
1.5

                }



                // Apply intensifier effect from the previous word if
applicable

                if (previous_word_is_intensifier) {

                    sentimentValue += sentimentValue * INTENSIFIER; //
Amplify by 0.293 for intensifiers

                    previous_word_is_intensifier = false; // Reset the flag
after applying
```

```
                }


                // Apply negation effect if active
                if (negation_active) {
                    sentimentValue *= NEGATION; // Negate the sentiment by
multiplying with -0.5
                }


                // Adjust sentiment for exclamation marks if present
                if (exclamation > 0) {
                    if (sentimentValue > 0) {
                        sentimentValue += (sentimentValue * EXCLAMATION *
exclamation); // Positive boost for positive words
                    } else {
                        sentimentValue -= (sentimentValue * EXCLAMATION *
exclamation); // Negative boost for negative words
                    }
                }
            }


            // Store the sentiment value in the scores array
            scores[index] = sentimentValue;
            sentimentSum += sentimentValue; // Accumulate total sentiment
```

```
value



            index++; // Move to the next index for storing scores



            //CHATGPT HELPED ME WITH THIS LOGIC

            // Deactivate negation if a conjunction or punctuation is
encountered

            if (strcmp(lowerToken, "and") == 0 || strcmp(lowerToken, "or")
== 0 || strcmp(lowerToken, ".") == 0) {

                negation_active = false; // Reset negation effect

            }



            // Reset the intensifier flag for the current word if it's not
an intensifier

            previous_word_is_intensifier = false;

        }



        // Get the next word token

        token = strtok(NULL, " \n\t\v\f\r,.");

    }



    //CHATGPT HELPED ME WITH THIS LOGIC

    // Sift sentiment scores into positive, negative, and neutral sums
```

```
    sift_sentiment_scores(scores, index, &pos_sum, &neg_sum, &neu_count);


    // Calculate the compound score for the sentence using normalization
    float compound = sentimentSum / sqrt(sentimentSum * sentimentSum + 15);


    // Normalize the positive, negative, and neutral scores based on total
components
    float total = pos_sum + fabs(neg_sum) + neu_count;
    if (total == 0.0) total = 1.0; // Prevent division by zero by setting
total to 1


    float pos = fabs(pos_sum / total); // Calculate positive proportion
    float neg = fabs(neg_sum / total); // Calculate negative proportion
    float neu = (float)neu_count / total; // Calculate neutral proportion


    // Return the sentiment analysis results as a struct
    SentimentResult result = { pos, neg, neu, compound };
    return result; // Return the SentimentResult struct containing all
calculated scores
}
```

Makefile:

```
# Compiler
```

```
CC = gcc


#Flags

CFLAGS = -g -Wall -Wno-unused-variable


# Source Files

SRC = main.c vaderSentiment.c


# Header Files

HEADERS = utility.h


# Output Binary

OUTPUT = vaderSentiment


# Build the executable

all: $(OUTPUT)


$(OUTPUT): $(SRC) $(HEADERS)

   $(CC) $(CFLAGS) $(SRC) -o $(OUTPUT)


# Clean the build directory

clean:
```

```
	rm -f $(OUTPUT)



# Check memory leaks using Valgrind

valgrind: $(OUTPUT)

	valgrind --leak-check=full --track-origins=yes ./$(OUTPUT)



.PHONY: all clean valgrind
```