Implementing VADER Sentiment Analysis in C

Yousha Munowar

McMaster University

MECHTRON 2MP3

Pedram Pasandide

November 15th 2024

Algorithms Implemented

In the project, there were a total of 4 files utilized, the make file was utilized to ensure the files are properly compiled and can be executed in one word as opposed to multiple lines of compiling. This defines compilation of certain source files and links them together. This file also cleans any build artifacts, such as object files.

The main file is the driving logic of the program and uses the functions form the vaderSentiment.c file to calculate the outputs of the sentences passed.

The utility.h file serves as a foundation of the files as it defines any constants, structures and prototypes necessary for managing the data and the file.

As finally for the core file of this project, the vaderSentiment.c file is a tailored version of sentiment analysis. The file begins with processing the input text which is tokenized to words and is then assessed with respect to the file containing predefined lexicons. This is then given a sentimental score and is calculated using the method within the assignment file. The compound score, and the percentage of positivity, negativity, and naturality the sentence contains.

Test Cases: Case Study with VADER developed in C

SENTENCE	COMPOUND	
	Model in C	Model in Python
VADER is smart, handsome, and funny.	0.8316	0.8316
VADER is smart, handsome, and funny!	0.8539	0.8439
VADER is very smart, handsome, and funny.	0.8518	0.8545
VADER is VERY SMART, handsome, and FUNNY.	0.9071	0.9227
VADER is VERY SMART, handsome, and FUNNY!!!	0.9417	0.9342
VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY!!!	0.9417	0.9469
VADER is not smart, handsome, nor funny.	-0.5994	-0.7424

At least it isn't a horrible book.	-0.5423	0.431
The plot was good, but the characters are uncompelling and the dialog is not great.	-0.1406	-0.7042
Make sure you :) or :D today!	0.8945	0.8633
Not bad at all	0.3071	0.431

Compiling and Running

To compile and run this program, the following can be done in the terminal:

1. Download all the files and ensure they are all in the same directory
2. Once done, go to the terminal and simply type "make", this should in turn run the program and execute the files

Appendix

Utility.h

```
#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",
};
```

```c
// 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>
```

```c
#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;

// Structure to hold the sentiment analysis results
typedef struct {
float pos;
float neg;
float neu;
float compound;
} SentimentResult;

// Function prototypes
WordData *read_lexicon_file(const char *filename, int *word_count); // Reads lexicon file
SentimentResult calculate_sentiment_score(const char *sentence, WordData *lexicon, int
word_count); // 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>
#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");
```

```
    // Handle file opening errors
    if (!file) {
    perror("Failed to open the file");
    return NULL;
    }

    int capacity = 100; // Starting capacity for the array
    WordData *words = malloc(capacity * sizeof(WordData));
    // Check if memory allocation was successful
    if (!words) {
    perror("Unable to allocate memory");
    fclose(file);
    return NULL;
    }

    char line[256]; // Buffer for reading lines
    *word_count = 0; // Initialize word count

    // Read the file line by line
    while (fgets(line, sizeof(line), file)) {
    if (*word_count == capacity) {
    capacity += 100; // Increase capacity in steps of 100
    WordData *new_block = realloc(words, capacity * sizeof(WordData));
    if (!new_block) {
    perror("Failed to reallocate memory");
    free(words);
    fclose(file);
    return NULL;
    }
    words = new_block;
    }

    // Extract data from the line into the words array
    sscanf(line, "%s %f %f", words[*word_count].word, &words[*word_count].value1,
&words[*word_count].value2);
    (*word_count)++; // Increment the count of words
    }

    fclose(file); // Close the file after reading
    return words; // Return the dynamically allocated array
    }
```

```
WordData find_data(WordData *data, char *word) {
int i = 0;
while (data[i].word[0] != '\0') { // Continue until an empty word is encountered
if (strcmp(data[i].word, word) == 0) {
return data[i]; // Return the matching WordData struct
}
i++; // Move to the next index
}

WordData nullData; // Prepare a null data struct if no match is found
nullData.word[0] = '\0'; // Set the first character of word to null character
return nullData; // Return the null data struct
}




// 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;
*neg_sum = 0.0;
*neu_count = 0;

for (int i = 0; i < count; i++) {
if (sentiments[i] > 0) {
*pos_sum += (sentiments[i] + 1); // Compensation for positive sentiment
} else if (sentiments[i] < 0) {
*neg_sum += (sentiments[i] - 1); // Compensation for negative sentiment
} else {
(*neu_count)++;
}
}
}




// 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 };
        int index = 0;
        float sentimentSum = 0.0;
        float pos_sum = 0.0, neg_sum = 0.0;
        int neu_count = 0;

        // Copy the sentence so we can tokenize it without modifying 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,.");

        // Flags to track if previous word was an intensifier or negation
        bool previous_word_is_intensifier = false;
        bool negation_active = false;

        // Loop through each token to analyze its sentiment
        while (token != NULL) {
        bool allCaps = true; // Variable to check if the word is in ALLCAPS
        int exclamation = 0; // Counter for exclamation marks

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

        for (int i = 0; lowerToken[i] != '\0'; i++) {
        // Check if the current character is lowercase (to decide if ALLCAPS)
        if (islower(lowerToken[i])) {
        allCaps = false;
        }
        // Convert character to lowercase
        lowerToken[i] = tolower(lowerToken[i]);

        // Count and limit exclamation marks
        if (lowerToken[i] == '!') {
        exclamation++;
        lowerToken[i] = '\0'; // Remove the exclamation mark
        if (exclamation > 3) exclamation = 3; // Limit to a max of 3
        }
        }
```

```
// Find the sentiment value of the word in the lexicon
WordData wordData = find_data(lexicon, lowerToken);

// Check if the token is an intensifier or negation
bool is_intensifier = false;
bool is_negation = false;

// Check if the current word is an intensifier or negation
for (int i = 0; i < POSITIVE_INTENSIFIERS_SIZE; i++) {
if (strcmp(lowerToken, positive_intensifiers[i]) == 0) {
is_intensifier = true;
break;
}
}

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

if (is_intensifier) {
// Set flag for the next word
previous_word_is_intensifier = true;
} else if (is_negation) {
// Activate negation effect for subsequent words
negation_active = true;
} else if (wordData.word[0] != '\0') {
// Process the sentiment value if the word is in the lexicon
float sentimentValue = wordData.value1;

// Apply ALLCAPS amplification if needed
if (allCaps) {
sentimentValue *= CAPS;
}

// Apply previous intensifier if applicable
if (previous_word_is_intensifier) {
sentimentValue += sentimentValue * INTENSIFIER;
previous_word_is_intensifier = false; // Reset flag
}

// Apply ongoing negation effect
```

```
if (negation_active) {
sentimentValue *= NEGATION;
}

// Adjust sentiment for exclamation marks (if any)
if (sentimentValue > 0) {
sentimentValue += (sentimentValue * EXCLAMATION * exclamation);
} else {
sentimentValue -= (sentimentValue * EXCLAMATION * exclamation);
}

// Store the sentiment value and add it to the cumulative sentiment sum
scores[index] = sentimentValue;
sentimentSum += sentimentValue;

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

// Deactivate negation if a punctuation or conjunction is reached
if (strcmp(lowerToken, "and") == 0 || strcmp(lowerToken, "or") == 0) {
negation_active = false;
}

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

// Use sift_sentiment_scores to categorize the individual word sentiment scores
sift_sentiment_scores(scores, index, &pos_sum, &neg_sum, &neu_count);

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

// Normalize positive, negative, and neutral scores based on total sentiment components
float total = pos_sum + fabs(neg_sum) + neu_count;
float pos = fabs(pos_sum / total);
float neg = fabs(neg_sum / total);
float neu = fabs((float)neu_count / total);

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

Main.c

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

int main() {
int word_count = 0;

// Read the lexicon file
WordData *lexicon = read_lexicon_file("vader_lexicon.txt", &word_count);

if (lexicon == NULL) {
printf("File not readable.\n");
return 1;
}

// 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
for (int i = 0; i < sizeof(test_sentences) / sizeof(test_sentences[0]); i++) {
SentimentResult sentiment_score = calculate_sentiment_score(test_sentences[i], lexicon,
word_count);
printf("Sentence: \"%s\"\n", test_sentences[i]);
printf("This sentence was %.0f%% negative, %.0f%% neutral, %.0f%% positive, and had a
compound score of %.4f.\n\n",
```

```
        sentiment_score.neg * 100, sentiment_score.neu * 100, sentiment_score.pos * 100,
sentiment_score.compound);
        }

        // Free the allocated memory
        free(lexicon);
        return 0;
        }
```

Makefile

```
        # Compiler to use
        CC=gcc

        # Compiler flags
        CFLAGS=-Wall -g

        # Executable name
        TARGET=SentimentAnalyzer

        # Source files
        SOURCES=main.c vaderSentiment.c

        # Object files
        OBJECTS=$(SOURCES:.c=.o)

        # Default target, default action is to build and run
        default: build run

        # Build the program
        build: $(TARGET)

        # Link the object files into the executable
        $(TARGET): $(OBJECTS)
        $(CC) $(CFLAGS) -o $@ $^

        # Compile source files into object files
        %.o: %.c
        $(CC) $(CFLAGS) -c $<

        # Run the program
        run:
        ./$(TARGET)
```

```
# Clean up
clean:
rm -f $(OBJECTS) $(TARGET)

# This target tells make that these are not file names.
.PHONY: default build run clean
```

References

Chatgpt. (2022, November 30). https://chatgpt.com/

For additional information on APA Style formatting, please consult the APA Style Manual, 7th Edition.