Assignment 3

Sentence	Compound Score in Python	Compound Score in C
VADER is smart, handsome, and funny.	0.8316	0.8316
VADER is smart, handsome, and funny!	0.8439	0.8439
VADER is very smart, handsome, and funny.	0.8545	0.8518
VADER is VERY SMART, handsome, and FUNNY.	0.9227	0.9071
VADER is VERY SMART, handsome, and FUNNY!!!	0.9342	0.9057
VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY!!!	0.9469	0.9057
VADER is not smart, handsome, nor funny.	0.1027	0.1027
At least it isn't a horrible book.	-0.5423	-0.5423
The plot was good, but the characters are uncompelling and the dialog is not great.	-0.7042	-0.1406
Make sure you :) or :D today!	0.8633	0.8356
Not bad at all	0.431	0.3071

To run the file download all the files and place hen in the same directory. The run the make file by typing "make" into the terminal. After that run the "./sentiment_analysis" file that was just created.

The code works by first breaking up the sentence into its words by looking at where the spaces are. Then using functions to analyze each word for each bonus/amplifier I then apply all bonuses in the calculate score function. This function returns the total score so this then needs to be passed to the calculated compound where the returned value is the VADER score.

Key files and their function:

- **vadersentient.c** – File that contains all the functions to complete the analysis of the sentences.

- **main.c** – Main file that specifies what sentences to get the score for. Also includes the printf function to display the results.

- **untility.h** – Header file that contains all the functions in the vadersentient.c file as well as the data structures and constants.

- **Makefile** - File to compile all the files together and to produce the final running file. It also has the added function of being able to clean of the directory as needed if any files get corrupted or have issues.

Disclaimer: This code was written with the help of ChatGPT. I used it to speed up the process of writing the code however the structure of the code and the thinking behind the processes and functions was thought up by me.

Appendix:

vadersentient.c:

```
#include "utility.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <math.h>


// Function to count exclamation marks at the end of a sentence

int count_sentence_exclamation_marks(const char *sentence) {

   int count = 0;

   int len = strlen(sentence);

   // Start from the last character and move backwards

   for (int i = len - 1; i >= 0 && sentence[i] == '!'; i--) {

      count++;

   }

   return count;

}


// Check if a word is a positive intensifier

int is_positive_intensifier(const char *word) {

   const char *positive_intensifiers[] = {"very", "extremely", "absolutely", "completely",
"incredibly", "really", "so", "totally", "particularly", "exceptionally", "remarkably", NULL};

   for (int i = 0; positive_intensifiers[i] != NULL; i++) {

      if (strcmp(word, positive_intensifiers[i]) == 0) {
```

```
            return 1;
        }
    }
    return 0;
}

// Check if a word is a negative intensifier
int is_negative_intensifier(const char *word) {
    const char *negative_intensifiers[] = {"barely", "hardly", "scarcely", "somewhat", "mildly", "slightly", "partially", "fairly", "pretty much", NULL};
    for (int i = 0; negative_intensifiers[i] != NULL; i++) {
        if (strcmp(word, negative_intensifiers[i]) == 0) {
            return 1;
        }
    }
    return 0;
}

// Check if a word is a negation
int is_negation(const char *word) {
    const char *negations[] = {"not", "isn't", "doesn't", "wasn't", "shouldn't", "won't", "cannot", "can't", "nor", "neither", "without", "lack", "missing", NULL};
    for (int i = 0; negations[i] != NULL; i++) {
        if (strcmp(word, negations[i]) == 0) {
            return 1;
        }
    }
```

```
    return 0;
}

// Function to populate the lexicon data from a file into an array of WordData structures
int load_lexicon(const char *filename, WordData lexicon[], int max_entries) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Failed to open lexicon file");
        return 0;
    }

    char line[MAX_STRING_LENGTH * 2];
    int count = 0;

    while (fgets(line, sizeof(line), file) && count < max_entries) {
        WordData entry = {0};
        if (sscanf(line, "%s %f %f", entry.word, &entry.value1, &entry.value2) == 3) {
            lexicon[count++] = entry;
        }
    }

    fclose(file);
    return count;
}

// Function to convert a string to lowercase
```

```
void to_lower(char *str) {
   for (int i = 0; str[i]; i++) {
      str[i] = tolower((unsigned char)str[i]);
   }
}

// Function to remove only periods, exclamations, and commas from a word
void remove_punctuation(char *word) {
   int i = 0, j = 0;
   while (word[i]) {
      // Keep characters that are alphabetic, digits, hyphens, or not ., !, or ,
      if (isalpha(word[i]) || word[i] == '-' || isdigit(word[i]) || (word[i] != '.' && word[i] != '!' &&
word[i] != ',')) {
         word[j++] = word[i];
      }
      i++;
   }
   word[j] = '\0'; // Null-terminate the modified string
}

// Function to check if a word exists in the lexicon and retrieve its sentiment score
float get_lexicon_score(const char *word, WordData lexicon[], int lexicon_size) {
   if (word == NULL)
      return 0.0;

   char cleaned_word[MAX_STRING_LENGTH];
```

```
    strncpy(cleaned_word, word, MAX_STRING_LENGTH - 1);
    cleaned_word[MAX_STRING_LENGTH - 1] = '\0';

    to_lower(cleaned_word);
    remove_punctuation(cleaned_word);

    for (int i = 0; i < lexicon_size; i++) {
        if (strcmp(lexicon[i].word, cleaned_word) == 0) {
            return lexicon[i].value1;
        }
    }
    return 0.0;
}

// Tokenize a sentence into words
char **tokenize_sentence(const char *sentence, int *word_count) {
    char **words = malloc(MAX_STRING_LENGTH * sizeof(char *));
    char sentence_copy[MAX_STRING_LENGTH * MAX_WORDS];
    strncpy(sentence_copy, sentence, sizeof(sentence_copy) - 1);
    sentence_copy[sizeof(sentence_copy) - 1] = '\0';

    char *token = strtok(sentence_copy, " ");
    *word_count = 0;

    while (token != NULL && *word_count < MAX_STRING_LENGTH) {
        words[*word_count] = strdup(token);
```

```
        (*word_count)++;
        token = strtok(NULL, " ");
    }
    return words;
}

// Function to check if a word is composed entirely of uppercase letters, marking it lowercase if it contains specific symbols
int is_all_caps(const char *word) {
    const char *lowercase_symbols = "<>/?':;[{}]-=+!@#$^&*()|"; // Symbols that indicate lowercase
    int i = 0;

    // Return 0 if the word is empty
    if (word[i] == '\0') {
        return 0;
    }

    // Check each character in the word
    while (word[i] != '\0') {
        // If the character is in lowercase_symbols, immediately return 0
        if (strchr(lowercase_symbols, word[i]) != NULL) {
            return 0; // Contains a symbol that indicates the word is not all caps
        }

        // If it's an alphabetic character, ensure it's uppercase
        if (isalpha((unsigned char)word[i]) && !isupper((unsigned char)word[i])) {
```

```
            return 0; // Not an uppercase letter, so it's not all caps
        }
        i++;
    }

    return 1; // All alphabetic characters are uppercase and no forbidden symbols were found
}

// Calculate compound score
float calculate_compound_score(float score_total) {
    float compound = score_total/sqrt(score_total*score_total + ALPHA);
    return compound;
}

// Main function to calculate sentiment score based on rules
float calculate_score(const char *sentence, WordData lexicon[], int lexicon_size) {
    int word_count;
    float total_score = 0.0;
    char **words = tokenize_sentence(sentence, &word_count);

    for (int i = 0; i < word_count; i++) {
        if (strlen(words[i]) == 0) continue; // Skip empty tokens

        // Check if the original word is in ALLCAPS before any modification
        int is_allcaps = is_all_caps(words[i]);
```

```
// Make a copy of the original word for lexicon lookup
char og_word[MAX_STRING_LENGTH];
strncpy(og_word, words[i], MAX_STRING_LENGTH - 1);
og_word[MAX_STRING_LENGTH - 1] = '\0'; // Null-terminate the copied string

// Lowercase and remove punctuation
to_lower(words[i]);
remove_punctuation(words[i]);

float score = get_lexicon_score(words[i], lexicon, lexicon_size);
float modifier = 1.0;

// Check and apply modifiers
if (i > 0 && is_positive_intensifier(words[i - 1])) {
   modifier *= (1 + BOOST_FACTOR);
}

if (i > 0 && is_negative_intensifier(words[i - 1])) {
   modifier *= (1 - BOOST_FACTOR);
}

if (i > 0 && is_negation(words[i - 1])) {
   modifier *= -0.5;
}
```

```
    // Apply ALLCAPS emphasis if the original word was in ALLCAPS and has a lexicon score
    if (is_allcaps && score != 0) {
        modifier *= CAPS_MULTIPLIER;
    }

    // Apply the modifier to the score
    float final_score = score * modifier;
    total_score += final_score;
  }

  // Add exclamation emphasis
  int exclamation_count = count_sentence_exclamation_marks(sentence);
  float exclamation_boost = exclamation_count * EXCLAMATION_BOOST;
  total_score += exclamation_boost;

  // Free allocated memory
  for (int i = 0; i < word_count; i++) {
    free(words[i]);
  }
  free(words);

  return total_score;
}
```

Utility.h:

```
#ifndef UTILITY_H
```

```
#define UTILITY_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

// Constants used for sentiment analysis scoring and memory allocation
#define BOOST_FACTOR 0.293
#define CAPS_MULTIPLIER 1.5
#define EXCLAMATION_BOOST 0.292
#define MAX_STRING_LENGTH 50
#define MAX_WORDS 10000
#define ALPHA 15

// Data structure for lexicon words
typedef struct {
   char word[MAX_STRING_LENGTH];
   float value1;
   float value2;
} WordData;

// Function declarations

// Count exclamation marks at the end of a sentence
```

```
int count_sentence_exclamation_marks(const char *sentence);

// Check if a word is a positive intensifier
int is_positive_intensifier(const char *word);

// Check if a word is a negative intensifier
int is_negative_intensifier(const char *word);

// Check if a word is a negation
int is_negation(const char *word);

// Populate the lexicon data from a file
int load_lexicon(const char *filename, WordData lexicon[], int max_entries);

// Convert a string to lowercase
void to_lower(char *str);

// Remove only periods, exclamations, and commas from a word
void remove_punctuation(char *word);

// Check if a word exists in the lexicon and retrieve its sentiment score
float get_lexicon_score(const char *word, WordData lexicon[], int lexicon_size);

// Tokenize a sentence into words
char **tokenize_sentence(const char *sentence, int *word_count);
```

```
// Calculate compound score
float calculate_compound_score(float score_total);

// Check if a word is composed entirely of uppercase letters
int is_all_caps(const char *word);

// Calculate sentiment score based on rules
float calculate_score(const char *sentence, WordData lexicon[], int lexicon_size);

#endif // UTILITY_H
```

Main.c:

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

int main() {
    const char *filename = "vader_lexicon.txt";  // Path to your lexicon file
    WordData lexicon[MAX_WORDS];
    int lexicon_size = load_lexicon(filename, lexicon, MAX_WORDS);

    if (lexicon_size == 0) {
        printf("Failed to load lexicon data.\n");
        return 1;
    }
```

```
// Array of sentences to process
const 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"
};
int num_sentences = sizeof(sentences) / sizeof(sentences[0]);

for (int i = 0; i < num_sentences; i++) {
    printf("\nProcessing sentence: \"%s\"\n", sentences[i]);

    // Calculate the total and compound score using the lexicon
    float total_score = calculate_score(sentences[i], lexicon, lexicon_size);
    float compound_score = calculate_compound_score(total_score);

    //printf("Total Score: %.6f\n", total_score);
    printf("Compound Score: %.4f\n", compound_score);
}
```

```
    return 0;
}
```

Makefile:

```
# Compiler and flags
CC = gcc
CFLAGS = -Wall -Wextra -std=c99
LDFLAGS = -lm

# Target executable
TARGET = sentiment_analysis

# Object files
OBJS = main.o vaderSentiment.o

# Build target
$(TARGET): $(OBJS)
    $(CC) $(OBJS) -o $(TARGET) $(LDFLAGS)

# Compile main.c
main.o: main.c utility.h
    $(CC) -c main.c -o main.o $(CFLAGS)

# Compile vaderSentiment.c
vaderSentiment.o: vaderSentiment.c utility.h
    $(CC) -c vaderSentiment.c -o vaderSentiment.o $(CFLAGS)
```

```
# Clean up object files and executable
clean:
    rm -f $(OBJS) $(TARGET)
```