# 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)