2MP3 Assignment 3

Garret Langlois

November 2024

1 Introduction

Sentiment analysis programs such as VADER (Valence Aware Dictionary and sEntiment Reasoner) are very important in everyday life as they help to classify information from social media, reviews, and customer feedback. VADER works by using a lexicon full of words and known sentiment scores, this makes it highly reliable in real world scenarios. Different modifiers and intensifiers can be included to further refine the accuracy of the model dependinng on captialization, punctuation, and certain key intensifier words

Programs such as these often offer a tradeoff between memory management, speed and accuracy. The program I have developed uses a hash table to make parsing the lexicon highly efficient. This, however, neccessitates the use of a very large hash table to prevent collision. The program I have developed is free of memory leaks and parses the lexicon with an O(n) time complexity.

2 Problem Statement

In this assignment, you will implement a simplified version of the VADER sentiment analysis tool in C. In the future, if you are interested to develop more sophisticated VADER, you can start from here. The VADER implementation involves reading a lexicon file, identifying sentiment- bearing words within a sentence, and applying specific rules to calculate sentiment scores. You will be introduced to handling words with intensity modifiers (e.g., "very"), punctuation, ALLCAPS emphasis, and negations, simulating how VADER processes sentiment in real-world text. Inputs and Outputs are as follows:

Input

1. A sentence string (e.g., "VADER is very smart, handsome, and funny.").
2. A lexicon file (vader lexicon.txt) containing words with sentiment scores and senti- ment distributions from human ratings.

Output

1. Positive score (pos)
2. Negative score (neg)
3. Neutral score (neu)
4. Compound score (compound) - representing the overall sentiment strength, normalized to range from -1 to 1.

3 Solution

My solution involves the development of many different unique functions that take on different roles when calcualtating the sentiment score of the function. I will discuss them in the order that they occur from what the main function is first called.

3.1 Parse Lexicon

The first function that is called in my program is the **parseLexicon()** function. This function is responsible for taking the lexicon and transforming it into a list of structures which can be indexed as a hash table. This is the affomentioned function:

```
void parseLexicon(bool verbose) {

    //Open the file
    FILE *file = fopen("vader_lexicon.txt", "r");

    //Check to see if the file opening failed
    if (file == NULL) {
        printf("Error opening file\n");
        exit(1);
    }

    //Iterate through the file and parse the data
    for (int i = 0; i < LEXICON_SIZE; i++) {

        //We will start by reading the data and saving the values in the line to temporary values which are defined below
        //Then the word will be hashed to determine the index locaiton in the hash table and it will be saved as a WordData struct

        char wordToHash[17];
        float meanSentiment;
        float standardDeviation;
        int intArray[10];

        //This the fscanf function that reads the data from the file
        //There might be a more concise way to accomplish this but this increases readability
        if (fscanf(file, "%16s %f %f [%d, %d, %d, %d, %d, %d, %d, %d, %d, %d]", wordToHash, &meanSentiment, &standardDeviation,
        &intArray[0], &intArray[1], &intArray[2], &intArray[3], &intArray[4], &intArray[5], &intArray[6], &intArray[7], &intArray[8], &intArray[9])) {

            //Hash the word to determine the index location
            unsigned long hashed_location = hash(wordToHash) % TABLE_SIZE;

            //This fixes collisions, worst case scenario is O(n) where n is the number of elements in the table
            if (lexicon[hashed_location] != NULL) {

                //This allows for all of the print statements to be controlled
                if (verbose) {
                    printf("Collision detected at index %lu \n", hashed_location);
                    printf("\n Performing linear probing...\n");
                }

                //The linear probing is done by incrementing the index by 1 and then taking the modulo of the table size
                while (lexicon[hashed_location] != NULL) {
                    hashed_location++;
                    hashed_location = hashed_location % TABLE_SIZE;
                }

                //We can print out the new hashed location
                if (verbose) {
                    printf("New hashed location: %lu\n", hashed_location);
                }
            }

            //Now, as previously stated, we can place all of the data into the hash table
            //This starts with allocating memory for the WordData struct and then copying the data into the struct
            lexicon[hashed_location] = malloc(sizeof(WordData));

            //Check to see if the memory allocation failed
            if (lexicon[hashed_location] == NULL) {
```

```
                printf("Memory allocation failed\n");
                exit(1);
            }

            //Copy the data into the struct
            strcpy(lexicon[hashed_location]->word, wordToHash);
            lexicon[hashed_location]->meanSentiment = meanSentiment;
            lexicon[hashed_location]->standardDeviation = standardDeviation;

            //This does not effect the time complexity as it is constant time with sizing the
                lexicon
            for (int j = 0; j < 10; j++) {
                lexicon[hashed_location]->intArray[j] = intArray[j];
            }

            if (verbose) {
                printf("lexicon[%lu] = %s\n", hashed_location, lexicon[hashed_location]->word);
            }

        }

        else {
            printf("Parsing failure, exiting program...");
            exit(1);

        }
    }
    fclose(file);
}
}
```

Listing 1: parseLexicon function in C

This function works by iterating through the lexicon, depending on its size, and creating word structs for every word.

The structs are constructed using the following definition:

```
typedef struct {
    char word[17];
    float meanSentiment;
    float standardDeviation;
    int intArray[10];
} WordData;
```

3.2 Hashing and Storing Words

To make the time complexity of parsing the lexicon **O(1)**, we can hash all of the entries of the lexicon when we first parse it. This can be seen in the above **parseLexicon()** function when the **hash()** function is called. The hashing function that is used in my code is the common **djb2** hashing function. This is defined as the following:

```
unsigned long hash(char *str) {
    unsigned long hash = 5381;
    int c;
    while ((c = *str++)) {
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    }
    return hash;
}
```

3.3 Sentiment Calculation

The most important part of the code is to calculate both the sentiment score for each word but take into account the context of the sentence to calculate the compound score the sentence. The full compound score calculation can be found in the code itself however I will highlight the individual word sentiment calculations below. This is the code for the individual word calculations:

```c
double sentimentCalculation(char* testWord, double *intensifierScore, double *negationScore) {

    //printf("Negation score: %f\n", *negationScore);

    //Create a dummy word to store the original word, this way when it is lowercased we still have an OG copy to check all caps against
    char dummyWord[100];
    strcpy(dummyWord, testWord);

    //printf("%s\n", dummyWord);

    //printf("%f\n", *negationScore);

    //Variable to store the score of the word
    double wordScore = 0;

    //Variable to store the number of exclamation points
    int exclamation_count = 0;

    //We first need to check if the word has any punctuation attached to it
    //If it does we need to remove it
    for (size_t i = 0; i < strlen(testWord); i++) {
        if (testWord[i] == '.' || testWord[i] == ',' || testWord[i] == '?' || testWord[i] == ';' || testWord[i] == ':') {
            testWord[i] = '\0';
        }
        //If it has an exclamation point we need to boost the score
        else if (testWord[i] == '!' && exclamation_count <= 3) {
            wordScore += PUNCTUTATION_BOOST;
            testWord[i] = '\0';
            exclamation_count++;
        }
    }

    //Get the word data from the lexicon, this is O(1)
    WordData* wordData = findWord(testWord);

    //If the word is in the lexicon we can calculate the sentiment
    if (wordData != NULL) {

        //Get the sentiment value from the word data
        double sentimentValue = wordData->meanSentiment;

        //Add the sentiment value to the word score, inclusive of any previous intensifiers
        wordScore += (*intensifierScore + 1) * sentimentValue * (*negationScore);

        //printf("Word: %s, Sentiment: %f\n", testWord, wordScore);

        *negationScore = 1;

        //Check if the word is all caps
        int isAllCaps = 1;
        for (size_t m = 0; m < strlen(dummyWord); m++) {

            //This was from ChatGPT
            if (dummyWord[m] < 65 || dummyWord[m] > 90) {
                isAllCaps = 0;
                break;
            }
        }

        //If the word is all caps we need to boost the score
        if (isAllCaps) {
            wordScore *= 1.5;
        }

        //Reset the intensifier score
        *intensifierScore = 0;

        //Return the word score
        return wordScore;
```

```
    }

    //Even if the word is not in the lexicon we can still check if it is an intensifier
   *intensifierScore = 0;

   //Convert the word to lowercase
   to_lowercase(testWord);

    //Check if the word is an intensifier
    for (int j = 0; j < positiveCount; j++) {
        if (strcmp(testWord, positiveIntensifiers[j]) == 0) {
            *intensifierScore = boostFactor;
        }
    }

    //Check if the word is a negative intensifier
    if(*intensifierScore == 0) {
        for (int k = 0; k < negativeCount; k++) {
            if (strcmp(testWord, negativeIntensifiers[k]) == 0) {
                *intensifierScore = boostFactor;
            }
        }
    }

    *negationScore = 1;

    //Check if the word is negated

    int isNegated = 0;
    for (int l = 0; l < negationCount; l++) {
        if (strcmp(testWord, negations[l]) == 0) {
            isNegated = 1;
            break;
        }
    }

    //If the word is negated we need to multiply the score by the negation constant, which is
        defined in the utility header
    if (isNegated) {
        *negationScore = -0.5;
    }

    //Return 0 if the word is not in the lexicon
    return 0;

}
```

This code is split up into different sections. If the word is in the lexicon the the word the word score is calculated based on the sentiment score that is assigned to it in the lexicon and any previous boosts from intensifiers or negations. Additionally, there is an ALL CAPS modifier that is applied both if the word is in the lexicon and not in the lexicon. If the word is not in the lexicon the value of the intensifier score pointer is set so that is can be included in the next word sentiment calculation. The same thing is done with the negation pointer.

4 Example Sentence Results

A list of example sentences were presented to be tested in the code and have their results returned and compared with the python library. Below are the tabulated results.

The key take away from these results is that the difference in lexicon sizes and the inclusion of more words and ways that they are handled leads more accurate results in the python model. However, for less complex sentence structures my code matches nearly 1:1. Thank you for reading!

Table 1: Case Study with VADER developed in C

Sentence	**Compound Model in C**	**Python Library**
VADER is smart, handsome, and funny.	0.832	0.8316
VADER is smart, handsome, and funny!	0.844	0.8439
VADER is very smart, handsome, and funny.	0.852	0.8545
VADER is VERY SMART, handsome, and FUNNY.	0.886	0.9227
VADER is VERY SMART, handsome, and FUNNY!!!	0.893	0.9342
VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY!!!	0.893	0.9469
VADER is not smart, handsome, nor funny.	0.103	-0.7424
At least it isn't a horrible book.	-0.542	-0.5423
The plot was good, but the characters are uncompelling and the dialog is not great.	-0.141	-0.7042
Make sure you :) or :D today!	0.318	0.8633
Not bad at all	0.307	0.431

The following is the code that is used to calculate the sentiment scores in python from the **nltk** library:

```
from nltk.sentiment import SentimentIntensityAnalyzer


sia = SentimentIntensityAnalyzer()


sentences = [
    "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 i s n 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"
]


print("Sentence | Compound Score")
print("---------------------------------")
for sentence in sentences:
    sentiment = sia.polarity_scores(sentence)
    compound_score = sentiment['compound']
    print(f"{sentence} | {compound_score}")
```

5 Instructions for Operation

Sentiment Analysis is a key aspect of everyday interactions on social media and the internet as a whole.
To run the code simply compile and execute the code my typing **make** into the terminal and running using, and run it using **./vaderSentiment**. To create the shared library for windows type **make** into the terminal.

A C Code

```
//Define the header which includes the functions and includes
```

```
#include "utility.h"

//Define the hashtable lexicon array
WordData* lexicon[TABLE_SIZE];

//Define the positive intensifiers from the list given in the assignment
char* positiveIntensifiers[positiveCount] = {"absolutely", "completely", "extremely", "really", "so", "totally", "very", "particulary", "exceptionally",
"incredibly", "remarkably"};

//Define the negative intensifiers from the list given in the assignment
char* negativeIntensifiers[negativeCount] = {"barely", "hardly", "scarcely", "somewhat", "mildly", "slightly", "partially",
"fairly", "pretty much"};

//Define the negations from the list given in the assignment
char* negations[negationCount] = {"not", "isn't", "doesn't", "wasn't", "shouldn't", "won't", "cannot", "can't",
"nor", "neither", "without", "lack", "missing"};

//To make the the time complexity of the hash table O(1) we can use the djb2 hash function
//This is a hash function that I found online that is very fast

//The hash function starts by accepting the word that we want to hash
unsigned long hash(char *str) {

    //This is the seed
    unsigned long hash = 5381;
    int c;

    //This is the bit manipulation for the characters to create a unique hash
    while ((c = *str++)) {
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    }

    //This returns the hash from the function call
    //It is a unsigned long because the function returns a large value
    //This will later be remedied by the modulo operator
    return hash;
}

//Now we define a function to parse the lexicon and convert all of the words in wordData
void parseLexicon(bool verbose) {

    //Open the file
    FILE *file = fopen("vader_lexicon.txt", "r");

    //Check to see if the file opening failed
    if (file == NULL) {
        printf("Error opening file\n");
        exit(1);
    }

    //We can programatically determine the size of the lexicon by reading the number of lines in it
    int lexiconSize = 0;
    while (!feof(file)) {
        char ch = fgetc(file);
        if (ch == '\n') {
            lexiconSize++;
        }
    }

    rewind(file);

    //printf("Lexicon size: %d\n", lexiconSize);

    //Iterate through the file and parse the data
    for (int i = 0; i < lexiconSize; i++) {

        //We will start by reading the data and saving the values in the line to temporary values
            which are defined below
```

```
        //Then the word will be hashed to determine the index locaiton in the hash table and it will be saved as a WordData struct

        char wordToHash[17];
        float meanSentiment;
        float standardDeviation;
        int intArray[10];

        //This the fscanf function that reads the data from the file
        //There might be a more concise way to accomplish this but this increases readability
        if (fscanf(file, "%16s %f %f [%d, %d, %d, %d, %d, %d, %d, %d, %d, %d]", wordToHash, &meanSentiment, &standardDeviation,
        &intArray[0], &intArray[1], &intArray[2], &intArray[3], &intArray[4], &intArray[5], &intArray[6], &intArray[7], &intArray[8], &intArray[9])) {

            //Hash the word to determine the index location
            unsigned long hashed_location = hash(wordToHash) % TABLE_SIZE;

            //This fixes collisions, worst case scenario is O(n) where n is the number of elements in the table
            if (lexicon[hashed_location] != NULL) {

                //This allows for all of the print statements to be controlled
                if (verbose) {
                    printf("Collision detected at index %lu \n", hashed_location);
                    printf("\n Performing linear probing...\n");
                }

                //The linear probing is done by incrementing the index by 1 and then taking the modulo of the table size
                while (lexicon[hashed_location] != NULL) {
                    hashed_location++;
                    hashed_location = hashed_location % TABLE_SIZE;
                }

                //We can print out the new hashed location
                if (verbose) {
                    printf("New hashed location: %lu\n", hashed_location);
                }
            }

            //Now, as previously stated, we can place all of the data into the hash table
            //This starts with allocating memory for the WordData struct and then copying the data into the struct
            lexicon[hashed_location] = malloc(sizeof(WordData));

            //Check to see if the memory allocation failed
            if (lexicon[hashed_location] == NULL) {
                printf("Memory allocation failed\n");
                exit(1);
            }

            //Copy the data into the struct
            strcpy(lexicon[hashed_location]->word, wordToHash);
            lexicon[hashed_location]->meanSentiment = meanSentiment;
            lexicon[hashed_location]->standardDeviation = standardDeviation;

            //This does not effect the time complexity as it is constant time with sizing the lexicon
            for (int j = 0; j < 10; j++) {
                lexicon[hashed_location]->intArray[j] = intArray[j];
            }

            if (verbose) {
                printf("lexicon[%lu] = %s\n", hashed_location, lexicon[hashed_location]->word);
            }

        }

        else {
            printf("Parsing failure, exiting program...");
            exit(1);
```

```
        }
    }
    fclose(file);
}

//This function frees the tokens that were created in the tokenization function
void freeTokens(char** tokens, int tokenCount) {
    //Iterate through the tokens and free them
    for (int i = 0; i < tokenCount; i++) {
        free(tokens[i]);
    }
    //Free the tokens array
    free(tokens);
}

//This function prints the token list
void printTokenList(char** list, int tokenCount) {

    printf("[");

    for (int i = 0; i < tokenCount-1; i++) {
        printf("%s,",list[i]);
    }

    printf("%s", list[tokenCount-1]);
    printf("]");
}

//Custom tokenization function that will require reallocation of memory
char** tokenization(char * sentence, int size, int* tokenCount) {

    //Defines some dummy values to be modified
    int tokenSize = 10;
    int totalTokenLengthCount = 0;

    //Allocate memory for the tokens
    char** tokens = (char **)malloc(tokenSize * sizeof(char*));

    //Check to see if the memory allocation failed
    if (tokens == NULL) {
        printf("Error in tokenization memory allocation");
        exit(1);
    }


    int index = 0;
    int tokenLength = 0;
    char delimeter = ' ';

    //Iterate through the sentence and create tokens
    while (sentence[index] != '\0') {

        //We need to check for delimeters and remove different parts of sentences
        if ((sentence[index] == delimeter || sentence[index] == ',') && tokenLength > 0) {

            //Allocate memory for the individual token
            totalTokenLengthCount += tokenLength;
            char* token = (char *)malloc((tokenLength + 1) * sizeof(char));

            //Check to see if the memory allocation failed
            if (token == NULL) {
                printf("Memory allocation error when creating individual tokens...");
                freeTokens(tokens, *tokenCount);
                exit(1);
            }

            //Copy the token into the allocated memory
            strncpy(token, &sentence[index - tokenLength], tokenLength);
            token[tokenLength] = '\0';

            //Remove spaces in token
```

```
            while (*token == ' ') {
                memmove(token, token + 1, strlen(token));
            }

            // Reallocate the tokens array if it runs out of space
            if (*tokenCount == tokenSize) {
                // Double the token array size so that there will always be enough space
                tokenSize *= 2;
                tokens = (char **)realloc(tokens, tokenSize * sizeof(char*));
                if (tokens == NULL) {
                    printf("Error in reallocating token array memory");
                    freeTokens(tokens, *tokenCount);
                    exit(1);
                }
            }

            //Add the token to the tokens array
            tokens[*tokenCount] = token;
            tokenLength = 0;

            //Increment the token count
            (*tokenCount)++;

            //Skip any extra spaces or commas after the token
            while (sentence[index] == ' ' || sentence[index] == ',') {
                index++;
            }
            continue; //Skip the regular index increment to avoid missing characters
        } else if (sentence[index] != ',') {
            tokenLength++;
        }
        index++;
    }

    //Handle the last token
    if (totalTokenLengthCount < size - 1) {

        //Allocate memory for the last token
        char* token = (char *)malloc((tokenLength + 1) * sizeof(char));
        if (token == NULL) {
            printf("Memory allocation error when creating the last token...");
            freeTokens(tokens, *tokenCount);
            exit(1);
        }

        //Copy the last token into the allocated memory
        strncpy(token, &sentence[index - tokenLength], tokenLength);
        token[tokenLength] = '\0';

        //Remove spaces in token
        while (*token == ' ') {
            memmove(token, token + 1, strlen(token));
        }

        tokens[*tokenCount] = token;
        (*tokenCount)++;
    }

    //Print and return the tokens
    //printTokenList(tokens, *tokenCount);
    return tokens;
}

//This function converts the word to lowercase
void to_lowercase(char *str) {
    for (int i = 0; str[i]; i++) {

        //This comes from one of the included headers
        str[i] = tolower(str[i]);
    }
}

```

```
//This function finds the word in the lexicon
WordData* findWord(char* word) {

    //Convert the word to lowercase because all of the words in the lexicon are lowercase
    to_lowercase(word);

    //Hash the word to determine the index location
    unsigned long tempWordPosition = hash(word) % TABLE_SIZE;

    //Print the hashed location if you want
    //printf("Hashed location: %lu\n", tempWordPosition);

    //Check the initial hashed location for a match
    if (lexicon[tempWordPosition] != NULL && strcmp(lexicon[tempWordPosition]->word, word) == 0) {
        return lexicon[tempWordPosition];
    }

    //Perform linear probing for a match within a set range if the initial hashed location does not match
    //There is a very low chance that the word will not be found within the first 10 checks
    int index = 0;
    while (index < 10) {

        //Increment position modulo the table size
        tempWordPosition = (tempWordPosition + 1) % TABLE_SIZE;

        //Check for match at the new position
        if (lexicon[tempWordPosition] != NULL && strcmp(lexicon[tempWordPosition]->word, word) == 0) {
            return lexicon[tempWordPosition];
        }

        index++;
    }

    //If word not found after probing, return NULL
    return NULL;
}

//This function calculates the sentiment of the word
double sentimentCalculation(char* testWord, double *intensifierScore, double *negationScore) {

    //printf("Negation score: %f\n", *negationScore);

    //Create a dummy word to store the original word, this way when it is lowercased we still have an OG copy to check all caps against
    char dummyWord[100];
    strcpy(dummyWord, testWord);

    //printf("%s\n", dummyWord);

    //printf("%f\n", *negationScore);

    //Variable to store the score of the word
    double wordScore = 0;

    //Variable to store the number of exclamation points
    int exclamation_count = 0;

    //We first need to check if the word has any punctuation attached to it
    //If it does we need to remove it
    for (size_t i = 0; i < strlen(testWord); i++) {
        if (testWord[i] == '.' || testWord[i] == ',' || testWord[i] == '?' || testWord[i] == ';' || testWord[i] == ':') {
            testWord[i] = '\0';
        }
        //If it has an exclamation point we need to boost the score
        else if (testWord[i] == '!' && exclamation_count <= 3) {
            wordScore += PUNCTUTATION_BOOST;
            testWord[i] = '\0';
            exclamation_count++;
        }
```

```
    }

    //Get the word data from the lexicon, this is O(1)
    WordData* wordData = findWord(testWord);

    //If the word is in the lexicon we can calculate the sentiment
    if (wordData != NULL) {

        //Get the sentiment value from the word data
        double sentimentValue = wordData->meanSentiment;

        //Add the sentiment value to the word score, inclusive of any previous intensifiers
        wordScore += (*intensifierScore + 1) * sentimentValue * (*negationScore);

        //printf("Word: %s, Sentiment: %f\n", testWord, wordScore);

        *negationScore = 1;

        //Check if the word is all caps
        int isAllCaps = 1;
        for (size_t m = 0; m < strlen(dummyWord); m++) {

            //This was from ChatGPT
            if (dummyWord[m] < 65 || dummyWord[m] > 90) {
                isAllCaps = 0;
                break;
            }
        }

        //If the word is all caps we need to boost the score
        if (isAllCaps) {
            wordScore *= 1.5;
        }

        //Reset the intensifier score
        *intensifierScore = 0;

        //Return the word score
        return wordScore;

    }

    //Even if the word is not in the lexicon we can still check if it is an intensifier
   *intensifierScore = 0;

   //Convert the word to lowercase
   to_lowercase(testWord);

    //Check if the word is an intensifier
    for (int j = 0; j < positiveCount; j++) {
        if (strcmp(testWord, positiveIntensifiers[j]) == 0) {
            *intensifierScore = boostFactor;
        }
    }

    //Check if the word is a negative intensifier
    if(*intensifierScore == 0) {
        for (int k = 0; k < negativeCount; k++) {
            if (strcmp(testWord, negativeIntensifiers[k]) == 0) {
                *intensifierScore = boostFactor;
            }
        }
    }

    *negationScore = 1;

    //Check if the word is negated

    int isNegated = 0;
    for (int l = 0; l < negationCount; l++) {
        if (strcmp(testWord, negations[l]) == 0) {
            isNegated = 1;
```

```
            break;
        }
    }

    //If the word is negated we need to multiply the score by the negation constant, which is
        defined in the utility header
    if (isNegated) {
        *negationScore = -0.5;
    }

    //Return 0 if the word is not in the lexicon
    return 0;

}

//This function calculates sentiment score for the whole sentence
double compoundSentimentScoreCalculation(char** tokens, int sentenceLength) {

    //Variable to store the intensifier score
    double intensifierScore = 0;

    double negationScore = 1;

    //Variable to store the total score
    double totalScore = 0;

    //Iterate through the tokens and calculate the individual sentiment scores
    for (int i = 0; i < sentenceLength; i++) {
        double addScore = sentimentCalculation(tokens[i], &intensifierScore, &negationScore);
        totalScore += addScore;
    }

    //Calculate the compound score
    double compoundScore = totalScore/sqrt(totalScore*totalScore +15);

    //Return the compound score
    return compoundScore;

}
```

Listing 2: C code for sorting algorithms

```
CC = gcc

CFLAGS = -O3 -Wall

TARGET = vader_sentiment

SRCS = main.c vaderSentiment.c

all: $(TARGET)

$(TARGET): $(SRCS)
    $(CC) $(CFLAGS) -o $(TARGET) $(SRCS)

clean:
    rm -f $(TARGET)
```

Listing 3: Makefile for compiling the sorting algorithms and creating the shared library

```
#include "utility.h"

int main(void) {
    bool excessPrinting = false;

    parseLexicon(excessPrinting);

    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 numSentences = sizeof(sentences) / sizeof(sentences[0]);

    for (int i = 0; i < numSentences; i++) {
        int tokenCount = 0;

        char* sentence = strdup(sentences[i]);  // Create a mutable copy of the sentence
        char** tokens = tokenization(sentence, strlen(sentence), &tokenCount);

        double compoundScore = compoundSentimentScoreCalculation(tokens, tokenCount);

        printf("Sentence: \"%s\"\n", sentences[i]);
        printf("Compound Sentiment Score: %.3f\n\n", compoundScore);

        freeTokens(tokens, tokenCount);
        free(sentence);
    }
    return 0;
}
```

Listing 4: C code containing the example sentences to be tested and for executing the initial functions

```
//The headers are included in this file
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdbool.h>
#include <ctype.h>

//Define everything that can be modified in the program
#define ARRAY_SIZE 10
#define MAX_STRING_LENGTH 17

//This needs to be edited
#define LEXICON_SIZE 7521
#define TABLE_SIZE 15000
#define PUNCTUTATION_BOOST 0.292
#define boostFactor 0.293
#define negationConstant -0.5
#define positiveCount 11
#define negativeCount 9
#define negationCount 13

//Define the WordData struct
typedef struct {
    char word[MAX_STRING_LENGTH];
    float meanSentiment;
    float standardDeviation;
    int intArray[ARRAY_SIZE];
} WordData;

//Define the "function prototypes"
extern WordData* lexicon[TABLE_SIZE];
void parseLexicon(bool verbose);
void freeTokens(char** tokens, int tokenCount);
void printTokenList(char** list, int tokenCount);
char** tokenization(char* sentence, int size, int* tokenCount);
WordData* findWord(char* word);
double sentimentCalculation(char* testWord, double *intensifierScore, double *negationScore);
double compoundSentimentScoreCalculation(char** tokens, int sentenceLength);
unsigned long hash(char* str);
```

Listing 5: C code header for the function defitions and the package headers