

# 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 depending on capitalization, 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, necessitates 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 sentiment 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 calculating 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 aforementioned function:

```
1 void parseLexicon(bool verbose) {
2
3     //Open the file
4     FILE *file = fopen("vader_lexicon.txt", "r");
5
6     //Check to see if the file opening failed
7     if (file == NULL) {
8         printf("Error opening file\n");
9         exit(1);
10    }
11
12    //Iterate through the file and parse the data
13    for (int i = 0; i < LEXICON_SIZE; i++) {
14
15        //We will start by reading the data and saving the values in the line to temporary values
16        //which are defined below
17        //Then the word will be hashed to determine the index location in the hash table and it
18        //will be saved as a WordData struct
19
20        char wordToHash[17];
21        float meanSentiment;
22        float standardDeviation;
23        int intArray[10];
24
25        //This is the fscanf function that reads the data from the file
26        //There might be a more concise way to accomplish this but this increases readability
27        if (fscanf(file, "%16s%f%f[%d,%d,%d,%d,%d,%d,%d,%d,%d,%d]", wordToHash, &
28            meanSentiment, &standardDeviation,
29            &intArray[0], &intArray[1], &intArray[2], &intArray[3], &intArray[4], &intArray[5], &
30            intArray[6], &intArray[7], &intArray[8], &intArray[9])) {
31
32            //Hash the word to determine the index location
33            unsigned long hashed_location = hash(wordToHash) % TABLE_SIZE;
34
35            //This fixes collisions, worst case scenario is O(n) where n is the number of elements
36            //in the table
37            if (lexicon[hashed_location] != NULL) {
38
39                //This allows for all of the print statements to be controlled
40                if (verbose) {
41                    printf("Collision detected at index %lu\n", hashed_location);
42                    printf("\nPerforming linear probing...\n");
43                }
44
45                //The linear probing is done by incrementing the index by 1 and then taking the
46                //modulo of the table size
47                while (lexicon[hashed_location] != NULL) {
48                    hashed_location++;
49                    hashed_location = hashed_location % TABLE_SIZE;
50                }
51
52                //We can print out the new hashed location
53                if (verbose) {
54                    printf("New hashed location: %lu\n", hashed_location);
55                }
56            }
57
58            //Now, as previously stated, we can place all of the data into the hash table
59            //This starts with allocating memory for the WordData struct and then copying the data
60            //into the struct
61            lexicon[hashed_location] = malloc(sizeof(WordData));
62
63            //Check to see if the memory allocation failed
64            if (lexicon[hashed_location] == NULL) {
```

```

58         printf("Memory allocation failed\n");
59         exit(1);
60     }
61
62     //Copy the data into the struct
63     strcpy(lexicon[hashed_location]->word, wordToHash);
64     lexicon[hashed_location]->meanSentiment = meanSentiment;
65     lexicon[hashed_location]->standardDeviation = standardDeviation;
66
67     //This does not effect the time complexity as it is constant time with sizing the
        lexicon
68     for (int j = 0; j < 10; j++) {
69         lexicon[hashed_location]->intArray[j] = intArray[j];
70     }
71
72     if (verbose) {
73         printf("lexicon[%lu]=\n", hashed_location, lexicon[hashed_location]->word);
74     }
75
76 }
77
78 else {
79     printf("Parsing failure, exiting program...");
80     exit(1);
81 }
82 }
83 }
84 fclose(file);
85 }
86 }

```

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:

```

1 typedef struct {
2     char word[17];
3     float meanSentiment;
4     float standardDeviation;
5     int intArray[10];
6 } 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:

```

1 unsigned long hash(char *str) {
2     unsigned long hash = 5381;
3     int c;
4     while ((c = *str++)) {
5         hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
6     }
7     return hash;
8 }

```

## 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:

```

1 double sentimentCalculation(char* testWord, double *intensifierScore, double *negationScore) {
2
3     //printf("Negation score: %f\n", *negationScore);
4
5     //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
6     char dummyWord[100];
7     strcpy(dummyWord, testWord);
8
9     //printf("%s\n", dummyWord);
10
11    //printf("%f\n", *negationScore);
12
13    //Variable to store the score of the word
14    double wordScore = 0;
15
16    //Variable to store the number of exclamation points
17    int exclamation_count = 0;
18
19    //We first need to check if the word has any punctuation attached to it
20    //If it does we need to remove it
21    for (size_t i = 0; i < strlen(testWord); i++) {
22        if (testWord[i] == '.' || testWord[i] == ',' || testWord[i] == '?' || testWord[i] == ';' ||
23            testWord[i] == ':') {
24            testWord[i] = '\0';
25        }
26        //If it has an exclamation point we need to boost the score
27        else if (testWord[i] == '!' && exclamation_count <= 3) {
28            wordScore += PUNCTUTATION_BOOST;
29            testWord[i] = '\0';
30            exclamation_count++;
31        }
32    }
33
34    //Get the word data from the lexicon, this is O(1)
35    WordData* wordData = findWord(testWord);
36
37    //If the word is in the lexicon we can calculate the sentiment
38    if (wordData != NULL) {
39        //Get the sentiment value from the word data
40        double sentimentValue = wordData->meanSentiment;
41
42        //Add the sentiment value to the word score, inclusive of any previous intensifiers
43        wordScore += (*intensifierScore + 1) * sentimentValue * (*negationScore);
44
45        //printf("Word: %s, Sentiment: %f\n", testWord, wordScore);
46
47        *negationScore = 1;
48
49        //Check if the word is all caps
50        int isAllCaps = 1;
51        for (size_t m = 0; m < strlen(dummyWord); m++) {
52            //This was from ChatGPT
53            if (dummyWord[m] < 65 || dummyWord[m] > 90) {
54                isAllCaps = 0;
55                break;
56            }
57        }
58
59        //If the word is all caps we need to boost the score
60        if (isAllCaps) {
61            wordScore *= 1.5;
62        }
63
64        //Reset the intensifier score
65        *intensifierScore = 0;
66
67        //Return the word score
68        return wordScore;
69    }

```

```

70     }
71
72
73     //Even if the word is not in the lexicon we can still check if it is an intensifier
74     *intensifierScore = 0;
75
76     //Convert the word to lowercase
77     to_lowercase(testWord);
78
79     //Check if the word is an intensifier
80     for (int j = 0; j < positiveCount; j++) {
81         if (strcmp(testWord, positiveIntensifiers[j]) == 0) {
82             *intensifierScore = boostFactor;
83         }
84     }
85
86     //Check if the word is a negative intensifier
87     if(*intensifierScore == 0) {
88         for (int k = 0; k < negativeCount; k++) {
89             if (strcmp(testWord, negativeIntensifiers[k]) == 0) {
90                 *intensifierScore = boostFactor;
91             }
92         }
93     }
94
95     *negationScore = 1;
96
97     //Check if the word is negated
98
99     int isNegated = 0;
100    for (int l = 0; l < negationCount; l++) {
101        if (strcmp(testWord, negations[l]) == 0) {
102            isNegated = 1;
103            break;
104        }
105    }
106
107    //If the word is negated we need to multiply the score by the negation constant, which is
    defined in the utility header
108    if (isNegated) {
109        *negationScore = -0.5;
110    }
111
112    //Return 0 if the word is not in the lexicon
113    return 0;
114
115 }

```

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 it 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 un compelling 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 **nlk** library:

```

1 from nltk.sentiment import SentimentIntensityAnalyzer
2
3
4 sia = SentimentIntensityAnalyzer()
5
6
7 sentences = [
8     "VADER is smart, handsome, and funny!",
9     "VADER is very smart, handsome, and funny.",
10    "VADER is VERY SMART, handsome, and FUNNY.",
11    "VADER is VERY SMART, handsome, and FUNNY!!!",
12    "VADER is VERY SMART, uber handsome, and FRIGGIN FUNNY!!!",
13    "VADER is not smart, handsome, nor funny.",
14    "At least it is n t a horrible book.",
15    "The plot was good, but the characters are un compelling and the dialog is not great.",
16    "Make sure you :) or :D today!",
17    "Not bad at all"
18 ]
19
20
21 print("Sentence | Compound Score")
22 print("-----")
23 for sentence in sentences:
24     sentiment = sia.polarity_scores(sentence)
25     compound_score = sentiment['compound']
26     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 by 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

```

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

```

```

2 #include "utility.h"
3
4 //Define the hashtable lexicon array
5 WordData* lexicon[TABLE_SIZE];
6
7 //Define the positive intensifiers from the list given in the assignment
8 char* positiveIntensifiers[positiveCount] = {"absolutely", "completely", "extremely", "really", "so",
9        ", "totally", "very", "particular", "exceptionally",
10 "incredibly", "remarkably"};
11
12 //Define the negative intensifiers from the list given in the assignment
13 char* negativeIntensifiers[negativeCount] = {"barely", "hardly", "scarcely", "somewhat", "mildly",
14        "slightly", "partially",
15 "fairly", "pretty_much"};
16
17 //Define the negations from the list given in the assignment
18 char* negations[negationCount] = {"not", "isn't", "doesn't", "wasn't", "shouldn't", "won't", "
19        cannot", "can't",
20 "nor", "neither", "without", "lack", "missing"};
21
22 //To make the the time complexity of the hash table O(1) we can use the djb2 hash function
23 //This is a hash function that I found online that is very fast
24
25 //The hash function starts by accepting the word that we want to hash
26 unsigned long hash(char *str) {
27
28     //This is the seed
29     unsigned long hash = 5381;
30     int c;
31
32     //This is the bit manipulation for the characters to create a unique hash
33     while ((c = *str++)) {
34         hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
35     }
36
37     //This returns the hash from the function call
38     //It is a unsigned long because the function returns a large value
39     //This will later be remedied by the modulo operator
40     return hash;
41 }
42
43 //Now we define a function to parse the lexicon and convert all of the words in wordData
44 void parseLexicon(bool verbose) {
45
46     //Open the file
47     FILE *file = fopen("vader_lexicon.txt", "r");
48
49     //Check to see if the file opening failed
50     if (file == NULL) {
51         printf("Error opening file\n");
52         exit(1);
53     }
54
55     //We can programatically determine the size of the lexicon by reading the number of lines in it
56     int lexiconSize = 0;
57     while (!feof(file)) {
58         char ch = fgetc(file);
59         if (ch == '\n') {
60             lexiconSize++;
61         }
62     }
63
64     rewind(file);
65
66     //printf("Lexicon size: %d\n", lexiconSize);
67
68     //Iterate through the file and parse the data
69     for (int i = 0; i < lexiconSize; i++) {
70
71         //We will start by reading the data and saving the values in the line to temporary values
72         //which are defined below

```

```

69 //Then the word will be hashed to determine the index locaiton in the hash table and it
    will be saved as a WordData struct
70
71 char wordToHash[17];
72 float meanSentiment;
73 float standardDeviation;
74 int intArray[10];
75
76 //This the fscanf function that reads the data from the file
77 //There might be a more concise way to accomplish this but this increases readability
78 if (fscanf(file, "%16s%f%f[%d,%d,%d,%d,%d,%d,%d,%d,%d,%d]", wordToHash, &
79 meanSentiment, &standardDeviation,
80 &intArray[0], &intArray[1], &intArray[2], &intArray[3], &intArray[4], &intArray[5], &
81 intArray[6], &intArray[7], &intArray[8], &intArray[9])) {
82
83 //Hash the word to determine the index location
84 unsigned long hashed_location = hash(wordToHash) % TABLE_SIZE;
85
86 //This fixes collisions, worst case scenario is O(n) where n is the number of elements
    in the table
87 if (lexicon[hashed_location] != NULL) {
88
89 //This allows for all of the print statements to be controlled
90 if (verbose) {
91 printf("Collision detected at index %lu\n", hashed_location);
92 printf("\nPerforming linear probing...\n");
93 }
94
95 //The linear probing is done by incrementing the index by 1 and then taking the
    modulo of the table size
96 while (lexicon[hashed_location] != NULL) {
97 hashed_location++;
98 hashed_location = hashed_location % TABLE_SIZE;
99 }
100
101 //We can print out the new hashed location
102 if (verbose) {
103 printf("New hashed location: %lu\n", hashed_location);
104 }
105
106 //Now, as previously stated, we can place all of the data into the hash table
107 //This starts with allocating memory for the WordData struct and then copying the data
    into the struct
108 lexicon[hashed_location] = malloc(sizeof(WordData));
109
110 //Check to see if the memory allocation failed
111 if (lexicon[hashed_location] == NULL) {
112 printf("Memory allocation failed\n");
113 exit(1);
114 }
115
116 //Copy the data into the struct
117 strcpy(lexicon[hashed_location]->word, wordToHash);
118 lexicon[hashed_location]->meanSentiment = meanSentiment;
119 lexicon[hashed_location]->standardDeviation = standardDeviation;
120
121 //This does not effect the time complexity as it is constant time with sizing the
    lexicon
122 for (int j = 0; j < 10; j++) {
123 lexicon[hashed_location]->intArray[j] = intArray[j];
124 }
125
126 if (verbose) {
127 printf("lexicon[%lu] = %s\n", hashed_location, lexicon[hashed_location]->word);
128 }
129 }
130
131 else {
132 printf("Parsing failure, exiting program...");
133 exit(1);

```



```

134     }
135 }
136 fclose(file);
137 }
138
139 //This function frees the tokens that were created in the tokenization function
140 void freeTokens(char** tokens, int tokenCount) {
141     //Iterate through the tokens and free them
142     for (int i = 0; i < tokenCount; i++) {
143         free(tokens[i]);
144     }
145     //Free the tokens array
146     free(tokens);
147 }
148
149 //This function prints the token list
150 void printTokenList(char** list, int tokenCount) {
151     printf("[");
152
153     for (int i = 0; i < tokenCount-1; i++) {
154         printf("%s,", list[i]);
155     }
156
157     printf("%s", list[tokenCount-1]);
158     printf("]");
159 }
160
161 //Custom tokenization function that will require reallocation of memory
162 char** tokenization(char * sentence, int size, int* tokenCount) {
163     //Defines some dummy values to be modified
164     int tokenSize = 10;
165     int totalTokenLengthCount = 0;
166
167     //Allocate memory for the tokens
168     char** tokens = (char **)malloc(tokenSize * sizeof(char*));
169
170     //Check to see if the memory allocation failed
171     if (tokens == NULL) {
172         printf("Error in tokenization memory allocation");
173         exit(1);
174     }
175
176     int index = 0;
177     int tokenLength = 0;
178     char delimiter = ' ';
179
180     //Iterate through the sentence and create tokens
181     while (sentence[index] != '\0') {
182         //We need to check for delimiters and remove different parts of sentences
183         if ((sentence[index] == delimiter || sentence[index] == ',') && tokenLength > 0) {
184             //Allocate memory for the individual token
185             totalTokenLengthCount += tokenLength;
186             char* token = (char *)malloc((tokenLength + 1) * sizeof(char));
187
188             //Check to see if the memory allocation failed
189             if (token == NULL) {
190                 printf("Memory allocation error when creating individual tokens...");
191                 freeTokens(tokens, *tokenCount);
192                 exit(1);
193             }
194
195             //Copy the token into the allocated memory
196             strncpy(token, &sentence[index - tokenLength], tokenLength);
197             token[tokenLength] = '\0';
198
199             //Remove spaces in token

```

```

206     while (*token == ' ') {
207         memmove(token, token + 1, strlen(token));
208     }
209
210     // Reallocate the tokens array if it runs out of space
211     if (*tokenCount == tokenSize) {
212         // Double the token array size so that there will always be enough space
213         tokenSize *= 2;
214         tokens = (char **)realloc(tokens, tokenSize * sizeof(char*));
215         if (tokens == NULL) {
216             printf("Error in reallocating token array memory");
217             freeTokens(tokens, *tokenCount);
218             exit(1);
219         }
220     }
221
222     // Add the token to the tokens array
223     tokens[*tokenCount] = token;
224     tokenLength = 0;
225
226     // Increment the token count
227     (*tokenCount)++;
228
229     // Skip any extra spaces or commas after the token
230     while (sentence[index] == ' ' || sentence[index] == ',') {
231         index++;
232     }
233     continue; // Skip the regular index increment to avoid missing characters
234 } else if (sentence[index] != ',') {
235     tokenLength++;
236 }
237 index++;
238 }
239
240 // Handle the last token
241 if (totalTokenLengthCount < size - 1) {
242
243     // Allocate memory for the last token
244     char* token = (char *)malloc((tokenLength + 1) * sizeof(char));
245     if (token == NULL) {
246         printf("Memory allocation error when creating the last token...");
247         freeTokens(tokens, *tokenCount);
248         exit(1);
249     }
250
251     // Copy the last token into the allocated memory
252     strncpy(token, &sentence[index - tokenLength], tokenLength);
253     token[tokenLength] = '\0';
254
255     // Remove spaces in token
256     while (*token == ' ') {
257         memmove(token, token + 1, strlen(token));
258     }
259
260     tokens[*tokenCount] = token;
261     (*tokenCount)++;
262 }
263
264 // Print and return the tokens
265 // printTokenList(tokens, *tokenCount);
266 return tokens;
267 }
268
269 // This function converts the word to lowercase
270 void to_lowercase(char *str) {
271     for (int i = 0; str[i]; i++) {
272
273         // This comes from one of the included headers
274         str[i] = tolower(str[i]);
275     }
276 }
277

```

```

278 //This function finds the word in the lexicon
279 WordData* findWord(char* word) {
280
281     //Convert the word to lowercase because all of the words in the lexicon are lowercase
282     to_lowercase(word);
283
284     //Hash the word to determine the index location
285     unsigned long tempWordPosition = hash(word) % TABLE_SIZE;
286
287     //Print the hashed location if you want
288     //printf("Hashed location: %lu\n", tempWordPosition);
289
290     //Check the initial hashed location for a match
291     if (lexicon[tempWordPosition] != NULL && strcmp(lexicon[tempWordPosition]->word, word) == 0) {
292         return lexicon[tempWordPosition];
293     }
294
295     //Perform linear probing for a match within a set range if the initial hashed location does not
296     //match
297     //There is a very low chance that the word will not be found within the first 10 checks
298     int index = 0;
299     while (index < 10) {
300         //Increment position modulo the table size
301         tempWordPosition = (tempWordPosition + 1) % TABLE_SIZE;
302
303         //Check for match at the new position
304         if (lexicon[tempWordPosition] != NULL && strcmp(lexicon[tempWordPosition]->word, word) ==
305             0) {
306             return lexicon[tempWordPosition];
307         }
308         index++;
309     }
310
311     //If word not found after probing, return NULL
312     return NULL;
313 }
314
315 //This function calculates the sentiment of the word
316 double sentimentCalculation(char* testWord, double *intensifierScore, double *negationScore) {
317
318     //printf("Negation score: %f\n", *negationScore);
319
320     //Create a dummy word to store the original word, this way when it is lowercased we still have
321     //an OG copy to check all caps against
322     char dummyWord[100];
323     strcpy(dummyWord, testWord);
324
325     //printf("%s\n", dummyWord);
326
327     //printf("%f\n", *negationScore);
328
329     //Variable to store the score of the word
330     double wordScore = 0;
331
332     //Variable to store the number of exclamation points
333     int exclamation_count = 0;
334
335     //We first need to check if the word has any punctuation attached to it
336     //If it does we need to remove it
337     for (size_t i = 0; i < strlen(testWord); i++) {
338         if (testWord[i] == '.' || testWord[i] == ',' || testWord[i] == '?' || testWord[i] == ';' ||
339             testWord[i] == ':') {
340             testWord[i] = '\0';
341         }
342         //If it has an exclamation point we need to boost the score
343         else if (testWord[i] == '!' && exclamation_count <= 3) {
344             wordScore += PUNCTUTATION_BOOST;
345             testWord[i] = '\0';
346             exclamation_count++;
347         }
348     }

```

```

346     }
347
348     //Get the word data from the lexicon, this is 0(1)
349     WordData* wordData = findWord(testWord);
350
351     //If the word is in the lexicon we can calculate the sentiment
352     if (wordData != NULL) {
353
354         //Get the sentiment value from the word data
355         double sentimentValue = wordData->meanSentiment;
356
357         //Add the sentiment value to the word score, inclusive of any previous intensifiers
358         wordScore += (*intensifierScore + 1) * sentimentValue * (*negationScore);
359
360         //printf("Word: %s, Sentiment: %f\n", testWord, wordScore);
361
362         *negationScore = 1;
363
364         //Check if the word is all caps
365         int isAllCaps = 1;
366         for (size_t m = 0; m < strlen(dummyWord); m++) {
367
368             //This was from ChatGPT
369             if (dummyWord[m] < 65 || dummyWord[m] > 90) {
370                 isAllCaps = 0;
371                 break;
372             }
373         }
374
375         //If the word is all caps we need to boost the score
376         if (isAllCaps) {
377             wordScore *= 1.5;
378         }
379
380         //Reset the intensifier score
381         *intensifierScore = 0;
382
383         //Return the word score
384         return wordScore;
385     }
386
387     //Even if the word is not in the lexicon we can still check if it is an intensifier
388     *intensifierScore = 0;
389
390     //Convert the word to lowercase
391     to_lowercase(testWord);
392
393     //Check if the word is an intensifier
394     for (int j = 0; j < positiveCount; j++) {
395         if (strcmp(testWord, positiveIntensifiers[j]) == 0) {
396             *intensifierScore = boostFactor;
397         }
398     }
399
400     //Check if the word is a negative intensifier
401     if(*intensifierScore == 0) {
402         for (int k = 0; k < negativeCount; k++) {
403             if (strcmp(testWord, negativeIntensifiers[k]) == 0) {
404                 *intensifierScore = boostFactor;
405             }
406         }
407     }
408 }
409
410 *negationScore = 1;
411
412 //Check if the word is negated
413
414 int isNegated = 0;
415 for (int l = 0; l < negationCount; l++) {
416     if (strcmp(testWord, negations[l]) == 0) {
417         isNegated = 1;

```

```

418         break;
419     }
420 }
421
422 //If the word is negated we need to multiply the score by the negation constant, which is
    defined in the utility header
423 if (isNegated) {
424     *negationScore = -0.5;
425 }
426
427 //Return 0 if the word is not in the lexicon
428 return 0;
429 }
430
431 //This function calculates sentiment score for the whole sentence
432 double compoundSentimentScoreCalculation(char** tokens, int sentenceLength) {
433     //Variable to store the intensifier score
434     double intensifierScore = 0;
435
436     double negationScore = 1;
437
438     //Variable to store the total score
439     double totalScore = 0;
440
441     //Iterate through the tokens and calculate the individual sentiment scores
442     for (int i = 0; i < sentenceLength; i++) {
443         double addScore = sentimentCalculation(tokens[i], &intensifierScore, &negationScore);
444         totalScore += addScore;
445     }
446
447     //Calculate the compound score
448     double compoundScore = totalScore/sqrt(totalScore*totalScore +15);
449
450     //Return the compound score
451     return compoundScore;
452 }
453
454 }
455

```

Listing 2: C code for sorting algorithms

```

1  CC = gcc
2
3  CFLAGS = -O3 -Wall
4
5  TARGET = vader_sentiment
6
7  SRCS = main.c vaderSentiment.c
8
9  all: $(TARGET)
10
11  $(TARGET): $(SRCS)
12      $(CC) $(CFLAGS) -o $(TARGET) $(SRCS)
13
14  clean:
15      rm -f $(TARGET)
16

```

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

```

1  #include "utility.h"
2
3  int main(void) {
4      bool excessPrinting = false;
5
6      parseLexicon(excessPrinting);
7
8      const char* sentences[] = {
9          "VADER is smart, handsome, and funny.",
10         "VADER is smart, handsome, and funny!",

```

```

11     "VADER_is_very_smart,_handsome,_and_funny.",
12     "VADER_is_VERY_SMART,_handsome,_and_FUNNY.",
13     "VADER_is_VERY_SMART,_handsome,_and_FUNNY!!!",
14     "VADER_is_VERY_SMART,_uber_handsome,_and_FRIGGIN_FUNNY!!!",
15     "VADER_is_not_smart,_handsome,_nor_funny.",
16     "At_least_it_isn't_a_horrible_book.",
17     "The_plot_was_good,_but_the_characters_are_uncompelling_and_the_dialog_is_not_great.",
18     "Make_sure_you:)_or:_D_today!",
19     "Not_bad_at_all"
20 };
21
22 int numSentences = sizeof(sentences) / sizeof(sentences[0]);
23
24 for (int i = 0; i < numSentences; i++) {
25     int tokenCount = 0;
26
27     char* sentence = strdup(sentences[i]); // Create a mutable copy of the sentence
28     char** tokens = tokenization(sentence, strlen(sentence), &tokenCount);
29
30     double compoundScore = compoundSentimentScoreCalculation(tokens, tokenCount);
31
32     printf("Sentence:_%s_\n", sentences[i]);
33     printf("Compound_Sentiment_Score:_.3f_\n", compoundScore);
34
35     freeTokens(tokens, tokenCount);
36     free(sentence);
37 }
38 return 0;
39 }

```

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

```

1 //The headers are included in this file
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <math.h>
6 #include <stdbool.h>
7 #include <ctype.h>
8
9 //Define everything that can be modified in the program
10 #define ARRAY_SIZE 10
11 #define MAX_STRING_LENGTH 17
12
13 //This needs to be edited
14 #define LEXICON_SIZE 7521
15 #define TABLE_SIZE 15000
16 #define PUNCTUTATION_BOOST 0.292
17 #define boostFactor 0.293
18 #define negationConstant -0.5
19 #define positiveCount 11
20 #define negativeCount 9
21 #define negationCount 13
22
23 //Define the WordData struct
24 typedef struct {
25     char word[MAX_STRING_LENGTH];
26     float meanSentiment;
27     float standardDeviation;
28     int intArray[ARRAY_SIZE];
29 } WordData;
30
31 //Define the "function prototypes"
32 extern WordData* lexicon[TABLE_SIZE];
33 void parseLexicon(bool verbose);
34 void freeTokens(char** tokens, int tokenCount);
35 void printTokenList(char** list, int tokenCount);
36 char** tokenization(char* sentence, int size, int* tokenCount);
37 WordData* findWord(char* word);
38 double sentimentCalculation(char* testWord, double *intensifierScore, double *negationScore);
39 double compoundSentimentScoreCalculation(char** tokens, int sentenceLength);
40 unsigned long hash(char* str);

```

---

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