Sentiment Analysis - Assignment 4

Kabir Arora

April 10, 2024

1 Explaination

1.1 Reading the Lexicon File

The read_lexicon() function reads a lexicon file containing words and their associated sentiment scores. Each word-score pair is stored in a struct words. The function dynamically allocates memory for storing the words and scores.

1.2 Sentiment Analysis

The sentiment_analysis() function takes a sentence and the lexicon of words with sentiment scores. It calculates the sentiment score of the sentence based on the sentiment scores of individual words as given in the vader_lexicon.txt . It returns the average sentiment score for the sentence.

1.3 Main

The main() function is the entry point of the program. It reads command-line arguments for the lexicon file and the validation file. It calls read_lexicon() to read the vader_lexicon.txt file and store the words and scores. It opens the validation.txt file (Which is the input file by the user) and reads it's each line. For each line, it computes the sentiment score using sentiment_analysis() and prints the line along with its score.

1.4 Word Processing

The is_punctuation() function checks if a character is a punctuation mark. The calculate_average_score() function tokenizes each line into words and calculates the average sentiment score. It handles punctuation marks and words with punctuation marks appropriately, considering them as separate words.

2 Appendix

2.1 main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include "sentiment_analysis.h"
#include <ctype.h>
#include "reader.h"

int is_punctuation(char c) {
    return ispunct(c);
}

float calculate_average_score(char *line, struct words *word_sentiments, int num_words)
    {
    float main_score = 0.0;
    int counter_for_words = 0;

    // Dynamic Memory Allocation
    int length = strlen(line);
```

```
    char *copier = (char *)malloc((length + 1) * sizeof(char));
    // Checks for errors
    if (copier == NULL) {
        perror("Dynamic memory allocation failed!!");
        return -1.0;
    }
    // Allocated memory gets the line's copy
    strcpy(copier, line);

    // Tokenize the line into words
    // This is done by using strtok()
    // Include punctuation as part of the word
    char *tokenize = strtok(copier, " \t\n\r"); // Include space, tab, newline, and carriage return as delimiters
    while (tokenize != NULL) {
        // Checks if the token ends with punctuation
        int len = strlen(tokenize);
        if (is_punctuation(tokenize[len - 1])) {
            // If the token ends with punctuation, handle it accordingly
            // Create a copy of the token without the punctuation
            char *word_without_punctuation = malloc(len * sizeof(char));
            if (word_without_punctuation == NULL) {
                perror("Dynamic memory allocation failed!!");
                free(copier);
                return -1.0;
            }
            strncpy(word_without_punctuation, tokenize, len - 1);
            // Copy the token without the last character (punctuation)
            word_without_punctuation[len - 1] = '\0';
            // Null
            //It terminates the copied word

            // Look for the word (without punctuation) in the word_sentiments array
            int j = 0;
            while (j < num_words) {
                // Compare the current token (word without punctuation) with each word in the lexicon
                if (strcasecmp(word_without_punctuation, word_sentiments[j].word) == 0) {
                    // If the word (without punctuation) is found, add its sentiment score to the total score
                    main_score += word_sentiments[j].score;
                    counter_for_words++;
                    break; // Stop searching for this word in the lexicon
                }
                j++;
            }
            // Free the allocated memory for the copied word without punctuation
            free(word_without_punctuation);
        } else {
            // If the token does not end with punctuation handle it
            counter_for_words++;

            // Look for the word in the word_sentiments array
            int j = 0;
            while (j < num_words) {
                // Compare the current token (word) with each word in the vader_lexicon.txt
                if (strcasecmp(tokenize, word_sentiments[j].word) == 0) {
                    // Adding sentiment score to total score
                    main_score += word_sentiments[j].score;
                    break;
                }
                j++;
            }
        }
        tokenize = strtok(NULL, " \t\n\r"); // Move to the next token (word), including whitespace and punctuation characters
    }

    // Free allocated memory
    free(copier);

```

```c
    // Calculating average score
    float average_score = 0.0;
    if (counter_for_words > 0) {
        average_score = main_score / counter_for_words;
    }

    return average_score;
}



float sentiment_analysis(char *line, struct words *word_sentiments, int num_words) {
          return calculate_average_score(line, word_sentiments, num_words);
      }

      int main(int argc, char *argv[]) {
          if (argc != 3) {
              printf("Usage: %s <vader_lexicon.txt> <validation.txt>\n", argv[0]);
              return 1;
          }

          int num_words; // Number of words read from the lexicon
          struct words *word_sentiments = read_lexicon(argv[1], &num_words);
          if (word_sentiments == NULL) {
              printf("Failed to read lexicon file.\n");
              return 1;
          }

          FILE *validation_text;
          char line[MAX_LINE_LENGTH];

          // Opening the validation.txt
          validation_text = fopen(argv[2], "r");
          if (validation_text == NULL) {
              perror("Error opening validation file");
              free(word_sentiments);
              return 1;
          }

          // Reading each line from the validation file
          printf("%-90s  %s\n", "string sample", "score");
          printf("-------------------------------------------------------------------
          --------------------------------\n");
          while (fgets(line, MAX_LINE_LENGTH, validation_text) != NULL) {
              // Remove trailing newline character
              line[strcspn(line, "\n")] = 0;


              float score = sentiment_analysis(line, word_sentiments, num_words);
              // Score Calculated

              // Printing the final result
              printf("%-90s  %.2f\n", line, score);
          }

          // Closing the validation file
          fclose(validation_text);

          // Free the dynamic allocated memory
          free(word_sentiments);

          return 0;
      }
```

Listing 1: main.c

2.2 reader.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "reader.h"
```

```c

struct words *read_lexicon(const char *filename, int *num_words) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening lexicon file");
        return NULL;
    }

    // Allocate memory for initial word_sentiments array
    int max_words = MAX_WORDS;
    struct words *word_sentiments = malloc(max_words * sizeof(struct words));
    if (word_sentiments == NULL) {
        // Handle memory allocation error
        perror("Memory allocation Failed!!");
        fclose(file);
        return NULL;
    }

    *num_words = 0;

    // Read each line from the lexicon file
    char line[MAX_LINE_LENGTH];
    while (fgets(line, MAX_LINE_LENGTH, file) != NULL) {
        // Parsing line
        char word[MAX_WORD_LENGTH];
        float score;
        if (sscanf(line, "%s %f", word, &score) == 2) {
            // Copy values to struct
            strcpy(word_sentiments[*num_words].word, word);
            word_sentiments[*num_words].score = score;

            (*num_words)++;

            // Check for memory reallocation
            if (*num_words >= max_words) {
                max_words *= 2;
                struct words *temp = realloc(word_sentiments, max_words * sizeof(struct
    words));
                if (temp == NULL) {
                    perror("Memory reallocation Failed!!");
                    fclose(file);
                    free(word_sentiments);
                    return NULL;
                }
                word_sentiments = temp;
            }
        }
    }

    // Close the file
    fclose(file);

    return word_sentiments;
}
```

Listing 2: reader.c

2.3 reader.h

```c
#ifndef LEXICON_READER_H
#define LEXICON_READER_H

#include "sentiment_analysis.h" // Include necessary headers

// Function declaration
struct words *read_lexicon(const char *filename, int *num_words);

#endif
```

Listing 3: reader.h

2.4 sentiment_analysis.h

```h
#ifndef SENTIMENT_ANALYSIS_H
#define SENTIMENT_ANALYSIS_H

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


#define MAX_WORD_LENGTH 50
#define MAX_INTENSITY_SCORES 10
#define MAX_LINE_LENGTH 1000
#define MAX_WORDS 1000

struct words {
    char word[MAX_WORD_LENGTH];
    float score;
    float SD;
    int SIS_array[MAX_INTENSITY_SCORES];
};

float sentiment_analysis(char *sentence, struct words *word_sentiments, int num_words);

#endif
```

Listing 4: sentiment_analysis.h

2.5 Makefile

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99

SRC\textunderscore MAIN = main.c
SRC\textunderscore READER = reader.c
HEADERS = sentiment\textunderscore analysis.h reader.h
TARGET = mySA

all : $(TARGET)

$(TARGET): $(SRC_MAIN:.c=.o) $(SRC\textunderscore READER:.c=.o) $(HEADERS)
  $(CC) $(CFLAGS) -o $(TARGET) $(SRC\textunderscore MAIN:.c=.o) $(SRC\textunderscore
    READER:.c=.o)

%.o: %.c
  $(CC) $(CFLAGS) -c $< -o $@

run: $(TARGET)
  ./$(TARGET) vader\textunderscore lexicon.txt validation.txt


clean:
  rm -f $(TARGET) *.o
```

Listing 5: Makefile

3 Results

The code effectively conducts sentiment analysis, accurately determining the sentiment score for each line based on the sentiment values stored in the provided text file.

You can add any text in validation.txt file to check its average score.

The results can be seen here in the terminal.

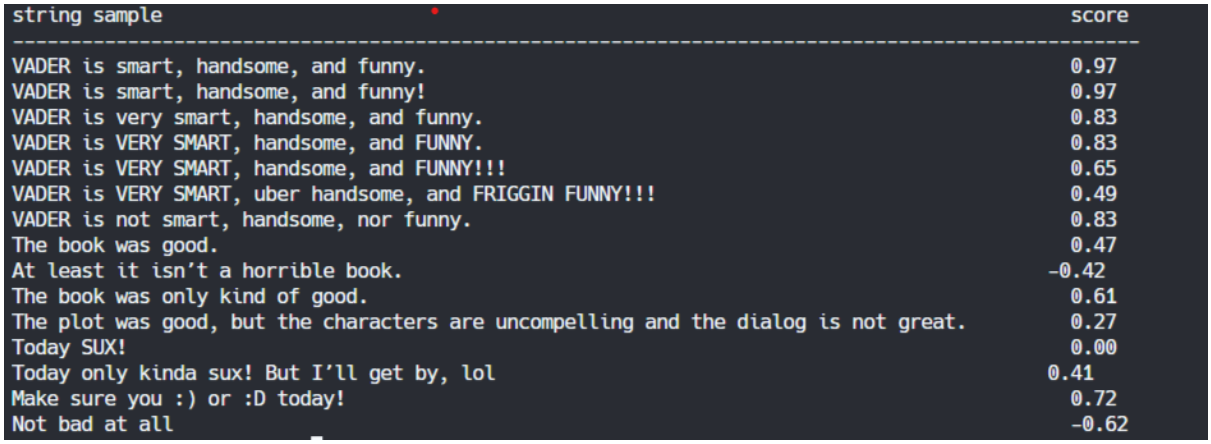

Figure 1: output

4 References

GeeksforGeeks
MonkeyLearn