COMPSCI 1XC3 - Assignment 4

Mina Al-Barak (ID: 400513160)

April 12, 2024

Chapter 1

Report Details

1.1 Program Description

The program utilizes different functions and calculations to obtain scores accurately. There were separate functions to calculate words and separate them from punctuation, such as "good,". Additionally, a separate array was designed to read emoji icons and distinguish them from words. The program combines basic programming concepts such as string length, arithmetic operations, and loops. These were crucial to read strings, distinguish words from emoji icons, and parse through all strings in a line. More complex concepts, such as dynamic memory allocation and timer, were used to efficiently store memory and hold memory blocks while processing lexicon arrays. The first function, **to_lower** converts strings into lowercase, so it can be read by the lexicon. Then, **compute_word_score** function deals with strings ending with punctuation by iterating through characters of the string and deleting the last character of the string until no punctuation marks are detected. This was an important function to integrate as, without it, the program can't process the score many words that are followed by punctuation marks and hence giving a score of 0. **is_emoticon** function checks for emoji icons in the line. It utilizes a hardcoded array of emoji icons and uses string comparison to check if any string/s in a line are emoji icons. **tokenize_string** function tokenizes sentences by breaking them down into smaller units, thereby allocating memory and adding to array. **compute_sentence_score** function takes strings and compares them to the lexicon scores to give them new scores in the array. It then calculates the average score for each sentence. Lastly, the **main** function defines the number of arguments, opens the lexicon and sentence files, calls the previous functions to parse the files, and prints the outputs.

1.2 Sources

I used a combination of Prof. Pasandide's lecture notes, as well as ChatGPT to help me implement the ctype.h library which I was unfamiliar with, in addition to aiding me in programming the **tokenize_string** function, and organizing the code overall.

Chapter 2

Appendix

2.1 mySA.c code

```c
#include <stdio.h> // contains printf(), fopen(), fclose(), fgets(), and sscanf().
#include <stdlib.h> // contains malloc(), realloc(), free(), and exit()
#include <string.h> // contains strlen(), strncpy(), strcmp(), strdup(), and strcspn()
#include <ctype.h> // contains isspace(), ispunct(), and tolower()
#include "mySA.h"

// structure stores the words and their sentiment score
struct words parse_line(const char *line)
{ //
    struct words new_word; // struct holds words and their new score
    new_word.word = malloc(256); // memory allocation to store words using malloc()
    if (new_word.word == NULL) // if memory allocation fails
    {
        printf("Memory allocation failed.");
        exit(1); // terminate program
    }

    // sscanf() scans sentence; it passes each string called 'line' and returns the score based on lexicon
    sscanf(line, "%s %f", new_word.word, &new_word.score); // %s is the string, %f is the floating-number score
    return new_word; //return new word & corresponding score based on lexicon
}

// this function converts strings to lowercase
void to_lower(char *str)
{
    for (int i = 0; str[i]; i++) // iterate over each character in the string
    {
        str[i] = tolower(str[i]); // tolower() converts uppercase letters to lowercase, so
    }                             // the string can be compared to the lexicon which is all lowercase
}

// this function deals with strings ending with punctuation; for example "good,", "FUNNY!!!"
```

```c
float compute_word_score(const char *word, struct words *lexicon, size_t lexicon_size)
{
    char *clean_word = strdup(word); // strdup() duplicates a string and allocates memory for it
    size_t len = strlen(clean_word); // strlen() finds string length
    while (len > 0 && ispunct(clean_word[len - 1])) // ispunct() checks if a character is punctuation
    {                                              // [len -1] is the last character (-1 due to indexing)
        clean_word[len - 1] = '\0'; // delete punctuation character by nullifying it
        len--; // continue shortening the string length until ispunct() can't find punctuation
    }

    // convert the clean word to lowercase
    for (size_t i = 0; i < len; i++) { // iterate over each character in the string
        clean_word[i] = tolower(clean_word[i]); // tolower() converts uppercase letters to lowercase
    }

    // search for clean word in lexicon
    float score = 0.0; // initializing score
    for (size_t i = 0; i < lexicon_size; ++i) // iterate over each word in the lexicon
    {
        if (strcmp(clean_word, lexicon[i].word) == 0) // strcmp() compares 2 strings to each other ie. it
        {                                             // checks if clean word matches anything in the lexicon
            score = lexicon[i].score; // score is now defined as what it is in the lexicon
            break;
        }
    }

    free(clean_word); // free memory
    return score;
}

// this function checks if we're dealing with emoji icons; for example ":D" (as to not confuse with punctuation)
int is_emoticon(const char *str)
{
    // made a list of emoticons to read as strings and not as punctuation
    const char *emoticons[] = {":)", ":(", ";)", ":D", ":P", ":-)", ":-(", ";-)", ":-D", ":-P"};
    int num_emoticons = sizeof(emoticons) / sizeof(emoticons[0]); // calculates number of emoji icons in array; good
                                                                   // for if I'd like to add more emojis to the array
    // check to see if any string matches an emoji icon
    for (int i = 0; i < num_emoticons; ++i) // iterate over each emoji icon
    {
        if (strcmp(str, emoticons[i]) == 0) // compare string to emoji icons to check if they match any emoji icon
        {
            return 1; // returns true if string is an emoji icon
        }
    }
    return 0; // otherwise return false
}

// this function tokenizes string ie. breaks down sentence into smaller parts to read words, punctuation & spaces
// typically five steps: find token length, allocate memory, copy token from string, nullify token, add to array
void tokenize_string(const char *str, char **tokens, int *num_tokens)
{
    int i = 0; // initialize index
    int token_start = -1; // initialize to -1 so it can identify when a new token begins (see below)
    while (str[i]) // iterate over each character in the string
    {
        if (!isspace(str[i])) // isspace() checks if character is a space; we're checking to see if it's
        {                         // NOT a space.
            if (token_start == -1) // checks if it's the start of a token
            {
                token_start = i; // start counting index of token
            }
        }
        else // otherwise, if it's NOT the start of token
        {
            if (token_start != -1)
            {
                int token_length = i - token_start; // calculate the token length
                char *token = malloc(token_length + 1); // allocate memory for token
                strncpy(token, &str[token_start], token_length); // strncpy() to copy token from the string
                token[token_length] = '\0'; // nullify token
                tokens[(*num_tokens)++] = token; // add token to array
                token_start = -1; // reset token start index
            }
            // check if character is a punctuation and the previous character is a space (indicating it's
            // the start of an emoji icon); for example: "happy :)" reads space before ":)"
            if (ispunct(str[i]) && (i == 0 || isspace(str[i - 1])))
            {
                int emoticon_length = 1; // set emoji icon length to 1 since they're considered single characters
                                         // by array we made earlier
                char *emoticon = malloc(emoticon_length + 1); // allocate memory for the emoji icon
                strncpy(emoticon, &str[i], emoticon_length); // copy emoji icon from the string
                emoticon[emoticon_length] = '\0'; // nullify eemoji icon
                if (is_emoticon(emoticon)) // check if it's an emoticon
                {
                    tokens[(*num_tokens)++] = emoticon; // add to array
                }
                else
                {
                    free(emoticon); // free memory
                }
            }
        }
        i++; // iterate through all the characters
    }
    if (token_start != -1) // if there's a token remaining
    {
        int token_length = i - token_start; // find token length
        char *token = malloc(token_length + 1); // allocate memory
        strncpy(token, &str[token_start], token_length); // copy token from string
        token[token_length] = '\0'; // nullify
        tokens[(*num_tokens)++] = token; // add to array
    }
}

// this function computes the sentiment score of a sentence
float compute_sentence_score(const char *sentence, struct words *lexicon, size_t lexicon_size)
{
    float sentence_score = 0.0; // initialize sentiment score
    int word_count = 0; // initialize word count of sentence

    // tokenize sentence using last function and calculate the scores for each word
    char *tokens[256]; // declaring array that stores the tokens
    int num_tokens = 0; // initialize token count
    tokenize_string(sentence, tokens, &num_tokens); // tokenize sentence
    for (int i = 0; i < num_tokens; i++) //iterate over each token
    {
        float word_score = compute_word_score(tokens[i], lexicon, lexicon_size); // compute score for word
        sentence_score += word_score; // update sentence score
        word_count++; // increment word count
        free(tokens[i]); // free allocated memory for token
    }

    if (word_count > 0) // checks if there are words in the sentence
    {
        sentence_score /= word_count; // average score calculator
    }
    return sentence_score;
}

// main function checks argument number, opens lexicon & sentences file, reads lexicon into memory, &
// computes sentiment score by processing sentences
int main(int argc, char *argv[])
{
    if (argc != 3) // checks if  number of command line arguments is not 3
    {
        printf("Usage: %s <lexicon_file> <sentences_file>\n", argv[0]); // print usage message
        return 1;
    }

    FILE *lexicon_file = fopen(argv[1], "r"); // Open lexicon file for reading; fopen() opens files
    if (lexicon_file == NULL) // if file doesn't open
    {
        printf("Can't open file."); // print error message
        return 1;
    }

    size_t lexicon_capacity = 100; // initialize lexicon capacity
    size_t lexicon_size = 0; // initialize size of lexicon
    struct words *lexicon = malloc(lexicon_capacity * sizeof(struct words)); // allocate memory for lexicon
    if (lexicon == NULL) // if memory allocation fails
    {
        printf("Memory allocation error."); // print error message
        fclose(lexicon_file); // close lexicon file using fclose()
        return 1;
    }

    char line[256]; // declare array to store lines from the file
    while (fgets(line, sizeof(line), lexicon_file)) // fgets() reads lines from a file
    {
        if (lexicon_size >= lexicon_capacity) // if lexicon capacity is exceeded
        {
            lexicon_capacity *= 2; // double the capacity
            struct words *temp = realloc(lexicon, lexicon_capacity * sizeof(struct words)); // reallocate memory
            if (temp == NULL) // if reallocation fails
            {
                printf("Memory allocation error"); // print error message
                fclose(lexicon_file); // close lexicon file
                free(lexicon); // free memory allocated for lexicon
                return 1;
            }
            lexicon = temp; // update lexicon timer
        }
        lexicon[lexicon_size++] = parse_line(line); // parse line and store word and score in lexicon
    }
    fclose(lexicon_file); // close lexicon file

    FILE *sentences_file = fopen(argv[2], "r"); // open sentences file for reading
    if (sentences_file == NULL) // if file doesn't opwn
    {
        printf("Can't open file."); // print error message
        free(lexicon); // free memory allocated for lexicon
        return 1;
    }

    // process sentences and compute scores
    printf("string\t\t\t\t\t\t\t\tsample\tscore\n"); // print header
    printf("-----------------------------------------------------------------------------\n"); // separator
    int sentence_number = 1; // initialize sentence number
    while (fgets(line, sizeof(line), sentences_file)) // reading each line in the sentences file
    {
        line[strcspn(line, "\n")] = 0; // remove processed line from further processing using strcspn() by
                                       // calculating initial string length & nullifying
        float score = compute_sentence_score(line, lexicon, lexicon_size); // compute sentence score
        printf("%-40s\t\t\t\t%.2f\n", line, score); // print sentence and score
        sentence_number++; // increment sentence number
    }

    fclose(sentences_file); // close sentences file
```

```c
    free(lexicon); // free memory allocated for lexicon

    return 0;
}
```

2.2 mySA.h code

```c
#ifndef MYSA_H
#define MYSA_H

struct words {
    char *word;
    float score;
    float SD;
    int SIS_array[10];
};

struct words parse_line(const char *line);

#endif /* MYSA_H */
```

2.3 Makefile

```makefile
CC = gcc
CFLAGS = -Wall -Wextra

all: mySA

mySA: mySA.c mySA.h

  $(CC) $(CFLAGS) -o mySA mySA.c

clean:
  rm -f mySA
```