Sentiment Analysis - Report

1XC3 Assignment #4

Matthew Zhou

April 9, 2024

Contents

1 Overview

The program preforms a rule-based sentiment analysis using the vader lexicon. It first parses the vader lexicon to find the scores of each token, then it finds and prints the average score of each line in a `txt` file using the lexicon.

1.1 Compiling

To compile the code, simply run the `make` command in the `src` directory. The object files and compiled program will be in a build directory. The compiled program will be named `mySA`. To clean the products of the build, run the `make clean` command in the `src` directory.

1.2 Running

When compiled the program takes two arguments `<lexicon-file>` and `<validation-file>`. The `<lexicon-file>` argument is the path to the vader lexicon. The `<validation-file>` argument is the path to a `.txt` file containing the strings to be analysed by the program.

2 Code Explanation

The code is split into two files. `parse_vader` contains functions to parse the `<lexicon-file>` and save all the words in a structure.

```
struct Words {
  char *word;
  float score;            // Polarity Score
  float SD;               // Standard Deviation
  int SIS_array[10];      // Sentiment Intensity Scores
};
```

`main` reads the content in `<validation-file>`, and uses the functions in the `parse_vader` to assign a score to each line in the `<validation-file>`.

2.1 `parse_vader`

The `parse_vader.h` file exposes the following functions:

```
// Frees a list of Words from memory
void freeWords(struct Words *words, int numWords);

// Reads a list of words from ‘file‘ and puts copies the result
   into ‘wordsPtr‘
// and ‘numWordsPtr‘. Caller is responsible for freeing ‘*wordsPtr
   ‘
// Returns EXIT_SUCCESS upon success and EXIT_FAILURE upon failure
int readWordsFromFile(FILE *file, struct Words **wordsPtr, int *
   numWordsPtr);

// Returns a pointer to a Words in a list ‘words‘ that matches ‘
   word‘
// Returns NULL if not found
struct Words *findWord(struct Words *words, int numWords, char *
   word);
```

2.1.1 `freeWords`

The `freeWords` function frees a dynamically allocated array of words.

```
void freeWords(struct Words *words, int numWords) {
  for (int i = 0; i < numWords; i++) {
    free(words[i].word);
  }
  free(words);
}
```

The function first iterates through each `Words` in the list of words, to free the `word` element. We do this because we use dynamic memory allocation to save the `word` element. We then free the list of words itself.

2.1.2 readWordsFromFile

The readWordsFromFile function parses a file with the same format as the vader_lexicon.txt file into a list of Words. The function updates the values referenced by wordsPtr and numWordsPtr to provide the list of words and the total number of words to the caller.

```
int readWordsFromFile(FILE *file, struct Words **wordsPtr, int *
    numWordsPtr) {
  // Allocate an array to store all the words
  struct Words *words = malloc(sizeof(struct Words));
  // Check the return value
  if (words == NULL) {
    fprintf(stderr, "Unable to allocate memory\n");
    return EXIT_FAILURE;
  }

  // Keep track of the number of words read
  int numWords = 0;

  int ret;
  // Keep parsing each line until there's an error
  do {
    struct Words word;
    ret = parseLine(file, &word);

    if (ret == EXIT_SUCCESS) {
      numWords++;
      words = realloc(words, sizeof(struct Words) * numWords);

      if (words == NULL) {
        fprintf(stderr, "Unable to allocate memory\n");
        return EXIT_FAILURE;
      }

      words[numWords - 1] = word;
    }
  } while (ret == EXIT_SUCCESS);

  // Hand the words and numWords to the caller
  *wordsPtr = words;
  *numWordsPtr = numWords;

  return EXIT_SUCCESS;
}
```

The function first allocates enough memory on the heap for one Words. If the function is unable to allocate the memory, it returns EXIT_FAILURE. It also initializes a count of the number of words read from the file (numWords).

It then tries to parse each line in the file, until there's an error. parseLine is a custom function that reads a line from file and parses it into the word struct. The function checks its return value to see if it is successful. If we are able to parse the line, we first increment the number of words counter. Then the function `realloc`s enough space for the new words list and appends the new word to the end of the list.

When the function is unable to parse a line from the file, we assume that we have hit the end of the file. We then hand the list of words and number of words back to the caller by mutating the value pointed at by `wordsPtr` and `numWordsPtr`.

2.1.3 `parseLine`

The `parseLine` function does the string parsing required to fill out the `Words` struct. It works as described in the above section.

```
// Parses a line from 'file' into a Words. Puts the output in '
    wordPtr'
// Returns EXIT_SUCESS upon success and EXIT_FAILURE on failure
int parseLine(FILE *file, struct Words *wordPtr) {
  // Read a line from file into buff
  char buff[LINE_SIZE];
  char *ret = fgets(buff, LINE_SIZE, file);

  if (ret == NULL) { // Check the return value
    return EXIT_FAILURE;
  }

  // Tokenize buff by \t

  // Read the word
  char *token = strtok(buff, "\t");
  if (token == NULL) {
    return EXIT_FAILURE;
  }

  char *word = malloc(strlen(buff) + 1);
  // Check the return value
  if (word == NULL) {
    fprintf(stderr, "Unable to allocate memory\n");
    return EXIT_FAILURE;
  }
  wordPtr->word = word;
  strcpy(wordPtr->word, token);

  // Read the score
  token = strtok(NULL, "\t");
  if (token == NULL) {
    return EXIT_FAILURE;
  }

  wordPtr->score = strtof(token, NULL);

  // Read the SD
  token = strtok(NULL, "\t");
  if (token == NULL) {
    return EXIT_FAILURE;
  }

  wordPtr->SD = strtof(token, NULL);

  // Read the SIS_array
  token = strtok(NULL, "\t");
  if (token == NULL) {
    return EXIT_FAILURE;
  }


  int scanRet = sscanf(token, "[%d, %d, %d, %d, %d, %d, %d, %d, %d
      , %d]", wordPtr->SIS_array, (wordPtr->SIS_array) + 1, (
      wordPtr->SIS_array) + 2, (wordPtr->SIS_array) + 3, (wordPtr->
      SIS_array) + 4, (wordPtr->SIS_array) + 5, (wordPtr->SIS_array
      ) + 6, (wordPtr->SIS_array) + 7, (wordPtr->SIS_array) + 8, (
      wordPtr->SIS_array) + 9);

  if (scanRet != 10) {
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

It first initializes a buffer on the stack to read the a line from the file. The buffer has max length of `LINE_SIZE`, which is defined in the `parse_vader.h` file to be 255. The function then reads a line from `file` into `buff`. If it is unsuccessful, it returns `EXIT_FAILURE`.

It then tokenizes the `buff` string by the tab character, since each section of the vader_lexicon is separated by a tab character. The function then parses each token in order to try and fill the `Words` struct.

To fill out the `word` element, it allocates enough memory for the word with the null character, and copies the token into the word element. If the function is unable to allocate enough memory, it returns `EXIT_FAILURE`.

To fill out the `score` and `SD` elements, it converts each token to a float, and writes the value into the structure.

To fill out the `SIS_array` element, the function uses `sscanf` to read each of the numbers in the formated token. It the checks the return value to see if the correct number of values were read.

2.1.4 `findWord`

The function looks in a list of `Words`s to find the `Words` with word element equal to `word`.

```
struct Words *findWord(struct Words *words, int numWords, char *
    word) {
  for (int i = 0; i < numWords; i++) {
```

```c
    if (strncmp(words[i].word, word, LINE_SIZE) == 0) {
      return words + i;
    }
  }

  return NULL;
}
```

It iterates through each word in the list, then stops and returns when it finds the right word. If it is unable to find the right word it returns `NULL`.

2.2 `main`

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

// Converts a null terminated string to lowercase
void strToLower(char *str) {
  for (int i = 0; str[i] != '\0'; i++) {
    if (isupper(str[i])) {
      str[i] = tolower(str[i]);
    }
  }
}

int main(int argc, char *argv[]) {
  // Check the number of arguments
  if (argc != 3) {
    printf("Usage ./mySA <lexicon_file> <validation_file> \n");
  }

  // Read the vader lexicon file
  FILE *vaderLexiconFile = fopen(argv[1], "r");
  if (vaderLexiconFile == NULL) {
    printf("Unable to open file\n");
    return EXIT_FAILURE;
  }

  // Read the files from the word
  int numWords = 0;
  struct Words *words;

  int ret = readWordsFromFile(vaderLexiconFile, &words, &numWords)
     ;
  if (ret == EXIT_FAILURE) {
    return EXIT_FAILURE;
  }

  // Read the file of stuff to analyze
  FILE *validationFile = fopen(argv[2], "r");
  if (validationFile == NULL) {
    printf("Unable to open file\n");
    return EXIT_FAILURE;
  }

  printf("%-100s %s\n", "string sample", "score");
  for (int i = 0; i < 120; i++) {
    printf("-");
  }
  printf("\n");

  // Read each line into a buffer
  char buff[255];
  while (fgets(buff, 255, validationFile) != NULL) {
    // Remove the trailing newline
    buff[strcspn(buff, "\n\r")] = 0;
    printf("%-100s", buff);

    float scoreSum = 0;
    int numTokens = 0;

    // Split the string by spaces and some punctuation
    // Note that this doesn't work in cases where "words"
    // have punctuation in it
    char *token = strtok(buff, " \r\n,.!");
    while (token != NULL) {
      numTokens++;

      // Find the word and add its score to the sum
      strToLower(token);
      struct Words *word = findWord(words, numWords, token);
      if (word != NULL) {
        scoreSum += word->score;
      }

      // Get the next token
      token = strtok(NULL, " \r\n,.!");
    }

    // Print the average score
    printf("%.5f\n", scoreSum / numTokens);
  }

  freeWords(words, numWords);
  fclose(vaderLexiconFile);
```

```
  fclose(validationFile);

  return EXIT_SUCCESS;
}
```

The strToLower function converts all uppercase characters in a string to lowercase. It iterates through the str until it encounters the null character. If the character at the current index is uppercase it converts it to lower.

The main funciton first checks the number of arguments. It then reads the file given by the first argument, and calls the readWordsFromFile function to get all the Words s. It puts the output into the variable words. If it is unable to open the file, it exits the program. The program then reads the file given by the second argument. If it is unable to open the file, it exits the program.

The program reads each line in the validationFile into buff. It then removes the traliing new line character and prints it out. It then tokenizes buff by spaces and punctuation, and searches the words list for the score of each token. After doing this for each token, it computes the average and prints it to the user.

At the very end of the program, we free all the memory used and close the files.

3 Appendix

```
// main.c
// Make clangd shut up
#define _CRT_SECURE_NO_WARNINGS

#include "parse_vader.h"
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Converts a null terminated string to lowercase
void strToLower(char *str) {
  for (int i = 0; str[i] != '\0'; i++) {
    if (isupper(str[i])) {
      str[i] = tolower(str[i]);
    }
  }
}

int main(int argc, char *argv[]) {
  // Check the number of arguments
  if (argc != 3) {
    printf("Usage ./mySA <lexicon_file> <validation_file> \n");
  }

  // Read the vader lexicon file
  FILE *vaderLexiconFile = fopen(argv[1], "r");
```

```
if (vaderLexiconFile == NULL) {
  printf("Unable to open file\n");
  return EXIT_FAILURE;
}

// Read the files from the word
int numWords = 0;
struct Words *words;

int ret = readWordsFromFile(vaderLexiconFile, &words, &numWords)
   ;
if (ret == EXIT_FAILURE) {
  return EXIT_FAILURE;
}

// Read the file of stuff to analyze
FILE *validationFile = fopen(argv[2], "r");
if (validationFile == NULL) {
  printf("Unable to open file\n");
  return EXIT_FAILURE;
}

printf("%-100s %s\n", "string sample", "score");
for (int i = 0; i < 120; i++) {
  printf("-");
}
printf("\n");

// Read each line into a buffer
char buff[255];
while (fgets(buff, 255, validationFile) != NULL) {
  // Remove the trailing newline
  buff[strcspn(buff, "\n\r")] = 0;
  printf("%-100s", buff);

  float scoreSum = 0;
  int numTokens = 0;

  // Split the string by spaces and some punctuation
  // Note that this doesn't work in cases where "words"
  // have punctuation in it
  char *token = strtok(buff, " \r\n,.!");
  while (token != NULL) {
    numTokens++;

    // Find the word and add its score to the sum
    strToLower(token);
    struct Words *word = findWord(words, numWords, token);
    if (word != NULL) {
        scoreSum += word->score;
      }

      // Get the next token
      token = strtok(NULL, " \r\n,.!");
    }

    // Print the average score
    printf("%.5f\n", scoreSum / numTokens);
  }

  freeWords(words, numWords);
  fclose(vaderLexiconFile);
  fclose(validationFile);

  return EXIT_SUCCESS;
}
```

```
// parse_vader.c
// Make clangd shut up
#define _CRT_SECURE_NO_WARNINGS

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

// Parses a line from `file` into a Words. Puts the output in `
   wordPtr`
// Returns EXIT_SUCESS upon success and EXIT_FAILURE on failure
int parseLine(FILE *file, struct Words *wordPtr) {
  // Read a line from file into buff
  char buff[LINE_SIZE];
  char *ret = fgets(buff, LINE_SIZE, file);

  if (ret == NULL) { // Check the return value
    return EXIT_FAILURE;
  }

  // Tokenize buff by \t

  // Read the word
  char *token = strtok(buff, "\t");
  if (token == NULL) {
    return EXIT_FAILURE;
  }

  char *word = malloc(strlen(buff) + 1);
  // Check the return value
  if (word == NULL) {
    fprintf(stderr, "Unable to allocate memory\n");
    return EXIT_FAILURE;
  }
  wordPtr->word = word;
  strcpy(wordPtr->word, token);

  // Read the score
  token = strtok(NULL, "\t");
  if (token == NULL) {
    return EXIT_FAILURE;
  }

  wordPtr->score = strtof(token, NULL);

  // Read the SD
  token = strtok(NULL, "\t");
  if (token == NULL) {
    return EXIT_FAILURE;
  }

  wordPtr->SD = strtof(token, NULL);

  // Read the SIS_array
  token = strtok(NULL, "\t");
  if (token == NULL) {
    return EXIT_FAILURE;
  }

  int scanRet = sscanf(token, "[%d, %d, %d, %d, %d, %d, %d, %d, %d
      , %d]",
                           wordPtr->SIS_array, (wordPtr->SIS_array) +
                              1,
                           (wordPtr->SIS_array) + 2, (wordPtr->
                              SIS_array) + 3,
                           (wordPtr->SIS_array) + 4, (wordPtr->
                              SIS_array) + 5,
                           (wordPtr->SIS_array) + 6, (wordPtr->
                              SIS_array) + 7,
                           (wordPtr->SIS_array) + 8, (wordPtr->
                              SIS_array) + 9);

  if (scanRet != 10) {
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

```c
void freeWords(struct Words *words, int numWords) {
  for (int i = 0; i < numWords; i++) {
    free(words[i].word);
  }
  free(words);
}

int readWordsFromFile(FILE *file, struct Words **wordsPtr, int *
    numWordsPtr) {
  // Allocate an array to store all the words
  struct Words *words = malloc(sizeof(struct Words));
  // Check the return value
  if (words == NULL) {
    fprintf(stderr, "Unable to allocate memory\n");
    return EXIT_FAILURE;
  }

  // Keep track of the number of words read
  int numWords = 0;

  int ret;
  // Keep parsing each line until there's an error
  do {
    struct Words word;
    ret = parseLine(file, &word);

    if (ret == EXIT_SUCCESS) {
      numWords++;
      words = realloc(words, sizeof(struct Words) * numWords);

      if (words == NULL) {
        fprintf(stderr, "Unable to allocate memory\n");
        return EXIT_FAILURE;
      }

      words[numWords - 1] = word;
    }
  } while (ret == EXIT_SUCCESS);

  // Hand the words and numWords to the caller
  *wordsPtr = words;
  *numWordsPtr = numWords;

  return EXIT_SUCCESS;
}

struct Words *findWord(struct Words *words, int numWords, char *
    word) {
  for (int i = 0; i < numWords; i++) {
    if (strncmp(words[i].word, word, LINE_SIZE) == 0) {
      return words + i;
    }
  }

  return NULL;
}
```

```
// parse_vader.h
#include <stdio.h>

#ifndef PARSE_VADER_H
#define PARSE_VADER_H

#define LINE_SIZE 255
#define SIS_ARRAY_SIZE 10

// Struct containing a word with its assosicated score and others
struct Words {
  char *word;
  float score;
  float SD;
  int SIS_array[SIS_ARRAY_SIZE];
};

// Frees a list of Words from memory
void freeWords(struct Words *words, int numWords);

// Reads a list of words from ‘file‘ and puts copies the result
    into ‘wordsPtr‘
// and ‘numWordsPtr‘. Caller is responsible for freeing ‘*wordsPtr
    ‘
// Returns EXIT_SUCCESS upon success and EXIT_FAILURE upon failure
int readWordsFromFile(FILE *file, struct Words **wordsPtr, int *
    numWordsPtr);

// Returns a pointer to a Words in a list ‘words‘ that matches ‘
    word‘
// Returns NULL if not found
struct Words *findWord(struct Words *words, int numWords, char *
    word);

#endif
```

```
# Makefile
CC = gcc
CFLAGS = -Wall -Wextra -g
BUILD_DIR = build
TARGET = mySA
```

```
all: $(BUILD_DIR)/$(TARGET)

$(BUILD_DIR)/$(TARGET): $(BUILD_DIR)/main.o $(BUILD_DIR)/
   parse_vader.o | $(BUILD_DIR)
        $(CC) $(CFLAGS) -o $(BUILD_DIR)/$(TARGET) $(BUILD_DIR)/
           parse_vader.o $(BUILD_DIR)/main.o

$(BUILD_DIR)/main.o: main.c parse_vader.h | $(BUILD_DIR)
        $(CC) $(CFLAGS) -o $@ -c main.c

$(BUILD_DIR)/parse_vader.o: parse_vader.c parse_vader.h | $(
   BUILD_DIR)
        $(CC) $(CFLAGS) -o $@ -c parse_vader.c

$(BUILD_DIR):
        mkdir $@

bruh:
        echo "bruh"

clean:
        rm -r $(BUILD_DIR)
```