Assignment 4 report

Yisi Liu

April 2024

1 Introduction

Computers are great at processing large amounts of data, but humans are great at understanding other humans. Getting computers to understand humans so they can process large amounts of people is a long-standing problem that many have sought to tackle in many different ways. Sentiment analysis targets the sentiment expressed in a sentence: whether it's positive, negative, and towards what is this attitude expressed.

There are two main types of sentiment analysis, machine-learning-based and rule-based. The former, obviously, relies on machine learning, while the latter relies on hard-coded rules. One very basic rule implemented in rule-based sentiment analysis is that the sentiment of a sentence can represented with a small positive or negative number that is the average of each word in the sentence's "sentiment score". VADER is a massive list of words with scores (and some other values for more advanced sentiment analysis) that is used like a dictionary. Words not in the dictionary are assumed to have a score of 0, expressing neither positive nor negative sentiment.

My code for assignment 4 is a very primitive implementation of rule-based sentiment analysis, using VADER and only coding the one rule described above.

2 The Results

My implementation is able to very basically identify the sentiment of a sentence that doesn't contain negatives (ex. "not happy" is scored positively because "not" has a score of 0), sarcasm, or other complications that could exploit the one implemented rule. However, the magnitude of the score doesn't scale with the magnitude of sentiment in a sentence because words such as "very" are scored 0 but add one to the total number of words, meaning a sentence like "I'm happy" will have a higher average than "I'm very happy" because the sum of each word's score is divided by 3 in the latter case. It also treats capitalized words the same as lowercase ones and does not account for the power of punctuation.

3 The Code

See my complete codes in Appendix 1.

First, the VADER lexicon is stored into an array of structs named "words" with dynamic memory allocation through file IO. A struct words has corresponding fields to everything on a line of a txt file containing VADER. Here, I wrote a void function "store" that takes in each piece of data that can be read each time with fscanf and a pointer to a struct words that simply puts everything in its proper place in the struct. I then use a while loop with said fscanf that dynamically allocates the space for another struct words in the array and calls store on the pointer to the latest space each time. The use of the while loop and the realloc to add space each time means my code is flexible to any length of file.

Second, I take the sentences I need to analyze through file IO and loop through my array of struct words containing all the data from the dictionary for each word. I have two arrays to help me do this. One is "scores", which keeps track of the scores of each sentence and is dynamically allocated and has a realloc inside a while loop just like above so it can take files containing however many number of sentences. The other is "tosum", which is reused on each sentence to temporarily keep track of the score of each word in a sentence so I can sum then average them before storing the average into scores and wiping tosum clean for the next sentence. I statically allocated 50 spots for tosum because an average human english sentence is 15-20 words long, with sentences of 40+ words being considered extremely long and extremely difficult if not grammatically incorrect. This does make my code less flexible but it's just so much less complex and practical enough anyway.

With another fscanf in a while loop, this time only scanning one word at a time and an fgetc in the loop that checks if the next character is space or a new line character, I'm able to iterate through the entire file containing all input sentences and differentiate between where one sentence ends and another begins. Before a sentence ends, I search my array of struct words for the word and search again with the punctuation removed and all capital letters turned into lowercase. When it ends, I sum tosum and put the average into scores. Credit to chatgpt for the function deleteChar, which deletes all instances of a character from a string.

4 Avenues of Improvement

Since this is a very primitive implementation of sentiment analysis, a lot of things can be done to instantly make it better.

First, the field in my struct words that keeps a string that is the word itself could be dynamically allocated according to the actual length of the word to use memory more precisely and probably defend against abnormally long words.

Second, as mentioned above, tosum could be dynamically allocated as well.

Third, I'm currently removing punctuation from words using deleteChar, but

a better way to do it is probably while the last character is not a letter, remove it, since the punctuation causing issues are periods, commas, exclamation marks, and the like. The current implementation also gets rid of apostrophes, which caused a small issue with it turning "I'll" into "ill" with the help of its capital-to-lowercase friend and incurring a negative score where there should be 0.

Fourth, now getting into the more unrealistic stuff for me currently, somehow keeping track of words that modify the sentiment of other words like "not" or "very "and applying their effect on the sentiment of a sentence to the sentence's score.

Fifth, doing the same keeping track and applying thing for capitalized words.

Sixth, accounting for punctuation.

5 Appendix 1

The following is my complete code.

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

struct words
{
    char word[30];//sorry to change it but this is just so much
    more convenient and i dont think there's a word longer than 30
    letters so i think it's fine
    float score;
    float SD;
    int SIS_array[10];
};

// put what i got from fscanf into struct words
void store(struct words *worddata, char *word, float score, float
    SD, int n1, int n2, int n3, int n4, int n5, int n6, int n7, int
     n8, int n9, int n10);

void deleteChar(char *str, char ch); // for sanitizing, from
    chatgpt

int main(int argc, char *argv[]) // first will be vader_lexicon,
    second will be validation
{

    // loop through file and store each line into an array of
    struct words

    // first make array
    struct words *arr = malloc(sizeof(struct words)); // can't use
    realloc before using malloc so here
    if (arr == NULL)                                    // check
    {
        printf("mem alloc failed. \n");
```

```
        return 1;
    }
    // then fill array
    // first open file
    FILE *lexicon = fopen(argv[1], "r");
    if (lexicon == NULL) // check
    {
        printf("could not open vader_lexicon.txt\n");
        return 1;
    }

    // now read file
    // variables for fscanf
    char word[30];
    float score;
    float SD;
    int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;

    int linenumber = 0;

    while (fscanf(lexicon, "%s  %f  %f  [%d, %d, %d, %d, %d, %d, %d
    , %d, %d, %d]\n", word, &score, &SD, &n1, &n2, &n3, &n4, &n5, &
    n6, &n7, &n8, &n9, &n10) != -1) // the emojis/phrases with
    spaces screw me up so i'll just ignore them ahaha...
    // for debugging printf("%s %f  %f  [%d, %d, %d, %d, %d, %d, %d
    , %d, %d, %d]\n", word, score, SD, n1, n2, n3, n4, n5, n6, n7,
    n8, n9, n10);
    {
        linenumber++;
        arr = realloc(arr, linenumber * sizeof(struct words)); //
    get space each time so it's flexy to sizey of filey
        store(arr + linenumber - 1, word, score, SD, n1, n2, n3, n4
    , n5, n6, n7, n8, n9, n10);
    }

    fclose(lexicon); // close vader_lexicon
    // also for debugging printf("%s\n", (arr+linenumber -1)->word)
    ;

    // loop through array of structs words to find score for each
    word in a sentence, calculate average for sentence

    // first, create array of all sentence scores for storage
    float *scores = malloc(sizeof(float)); // REMEMBER TO CHANGE TO
     DYM AND ADD REALLOC BELOW
    int linedex = -1;

    // next, create reusable array to keep track of all the scores
    of words in a sentence before averaging
    float tosum[50]; // human sentences are not very long plus this
     is for reusing, so i chose stack
    // initialize; zero unless found:
    for (int i = 0; i < 50; i++)
    {
        tosum[i] = 0;
    }
    int index = -1;
```

```
75
76      // next, go through file
77      // first, open validation to read
78      FILE *validation = fopen(argv[2], "r");
79
80      /*
81      while (fscanf(validation, "%s", word) != -1)
82      {
83          printf("%s", word);
84          if(fgetc(validation)=='\n')
85          {
86              printf("\n");
87          }
88      }*///tested structure in test.c!
89
90      char valword[30];
91
92      while (fscanf(validation, "%s", valword) != -1)
93      {
94          index++; // go to next element in array for each word
95          // printf("word: %s \n", valword);
96          // printf("struct: %s\n", (arr + 137)->word);
97          int unmatched = 1;
98          for (int i = 0; i < linenumber; i++)
99          {
100             if (strcmp((arr + i)->word, valword) == 0)
101             {
102                 tosum[index] = (arr + i)->score;
103                 unmatched = 0;
104                 // from debugging printf("matched first %s\n",
    valword);
105                 break;
106             }
107         }
108         if (unmatched) //if unmatched, scrub and try again (
    probably should've put the rest in some functions honestly but
    eh)
109         {
110             int l = strlen(valword);//for using in loops
111             for (int i = 0; i < l; i++)
112             {
113                 if (isupper(valword[i])) // no capitals
114                 {
115                     valword[i] = tolower(valword[i]);
116                 }
117             }
118             for (int i = 0; i < l; i++)
119             {
120                 if (isalpha(valword[i]) == 0) // no punctuation
121                 {
122                     deleteChar(valword, valword[i]);
123                     l = strlen(valword);
124                 }
125             }
126
127             // check again
128             for (int i = 0; i < linenumber; i++)
```

```
            {
                if (strcmp((arr + i)->word, valword) == 0)
                {
                    tosum[index] = (arr + i)->score;
                    // from debugging printf("matched clean %s\n",
valword);
                    break;
                }
            }
        }

        if (fgetc(validation) == '\n') // when gotten to the end of
 one line:
        {
            // a thing in scores = sum of tosum
            linedex++;
            scores = realloc(scores, (linedex + 1) * sizeof(float))
; // get space each time so it's flexy to sizey of filey again

            float sum = 0;
            for (int i = 0; i <= index; i++)
            {
                sum += tosum[i];
            }// from debugging printf("sum: %f, average: %f\n", sum
, sum / (index + 1));

            scores[linedex] = (sum / (index + 1));

            // reset for reuse
            for (int i = 0; i < 50; i++)
            {
                tosum[i] = 0;
            }
            index = -1;
        }
    }

    fclose(validation); // close file

    //okay wait i still need to print all the sentences but i don't
 wanna touch any code above here so ill just reopen and reclose
 validation

    FILE *val = fopen(argv[2], "r");

    int i = -1;
    char pad[100];

    while (fgets(pad, sizeof(pad), val) != NULL)
    {
        i++;
        deleteChar(pad, '\n');
        printf("%-100s", pad);
        printf("%7.2f\n", scores[i]);
    }

    fclose(val);
```

```
180
181
182     // free scores
183     free(scores);
184
185     // free dynamically allocated array of struct words
186     free(arr);
187
188     return 0;
189 }
190
191 //function implementations:
192
193 void store(struct words *worddata, char *word, float score, float
        SD, int n1, int n2, int n3, int n4, int n5, int n6, int n7, int
         n8, int n9, int n10)
194 {
195     worddata->score = score;
196     worddata->SD = SD;
197     worddata->SIS_array[0] = n1; // is there a way to do this that'
        s not manually stating each thing
198     worddata->SIS_array[1] = n2;
199     worddata->SIS_array[2] = n3;
200     worddata->SIS_array[3] = n4;
201     worddata->SIS_array[4] = n5;
202     worddata->SIS_array[5] = n6;
203     worddata->SIS_array[6] = n7;
204     worddata->SIS_array[7] = n8;
205     worddata->SIS_array[8] = n9;
206     worddata->SIS_array[9] = n10;
207     strcpy(worddata->word, word);
208 }
209
210 void deleteChar(char *str, char ch) //credit: chatgpt
211 {
212     int i, j;
213     int len = strlen(str);
214
215     // Iterate through the string
216     for (i = 0, j = 0; i < len; i++)
217     {
218         // If the current character is not the character to be
        deleted
219         if (str[i] != ch)
220         {
221             // Move the character to the left
222             str[j++] = str[i];
223         }
224     }
225     // Null-terminate the string after removing the character
226     str[j] = '\0';
227 }
```

Listing 1: main.c