20111051 Jaewoo An

Bioengineering Laboratory I

April 9, 2013

Laboratory 4: Text Mining

1. Source code with brief explanation

#include <iostream>

#include <fstream>

#include <string> // getline()

#include <math.h>

#include <sstream> // istringstream

using namespace std;

// Declaration of grobal variables //

int nTotalLineIndex = 0;

int nTheNumberofPMID = 0;

// Lower case function //

void toLowerCaseSTD(std::string&s)

{

for (std::string::iterator i = s.begin(); i != s.end(); ++i)

\*i = tolower(\*i);

}

// Read file //

ifstream readData(string strFileName)

{

ifstream f\_data(strFileName);

if (f\_data.is\_open()){ // f\_filenaem.is\_open() : check if a file stream was successful opening a file

cout <<"Reading data success!!"<< endl;

return f\_data;

}

else{

cout<<"Reading data is failed!!"<<endl;

exit(0);

}

}

string readLine(ifstream &DataName)

{

string strLine\_local;

getline(DataName,strLine\_local); // getline(f\_fileaname, stirng var) : extract characters from f\_filename data by line

toLowerCaseSTD(strLine\_local);

return strLine\_local;

}

/\*My code from here\*/

int countWords(string s) //NEW FUNCTION made for counting total tokens in a line of the literature.

{

int word\_count(0);

stringstream ss(s);

string word;

while(ss >> word) ++word\_count;

return word\_count;

}

/\*.............to here\*/

void chekingProcess(int nTheNumberofPMID)

{

if (nTheNumberofPMID % 1 == 0) cout << "The number of processed PMID : "<< nTheNumberofPMID<<endl;

}

string PMIDExtractor(string strLine, string strPreviousPMID)

{

int nPMIDChecker = strLine.find("pmid");

string strLocalPMID;

if (nPMIDChecker == 0){

strLocalPMID = strLine;

nTheNumberofPMID++;

chekingProcess(nTheNumberofPMID);

cout<<strLocalPMID<<endl;

}

else strLocalPMID = strPreviousPMID;

return strLocalPMID;

}

string manipulateGeneName(string strGeneToken)

{

strGeneToken.insert(strGeneToken.begin(),1,' ');

strGeneToken.insert(strGeneToken.end(),1,' ');

return strGeneToken;

}

int geneToeknMatching(istringstream &issGeneName, string strGeneToken, int nStandardGeneCheker, string strPubMedLine, ofstream &f\_Results,string strPMID)

{

string strStandardGeneID;

int tokencount = 0; //necessary to reset tokencount to zero for each line because when each line of PubMed is read, the tokencount starts from zero/////////////////////

while (getline(issGeneName,strGeneToken,'\t')){

// Searching standard gene //

nStandardGeneCheker++;

if (nStandardGeneCheker == 1)

{

strStandardGeneID = strGeneToken;

}

strGeneToken = manipulateGeneName(strGeneToken); // manipulating gene name for exact matching. ex) 'pa010' ==> ' pa010 ' //

// Matching genes in PubMed sentence //

if (int(strPubMedLine.find(strGeneToken)) != -1 && (strGeneToken.length() > 5)){ // string.find("word") : return position of first matched word, if there exists matching event to word user definded in string, otherwise return -1

int nCompareState = 0;

string strSentenceToken;

/\*my codes from here\*/ //this part of the code counts words in a line until GeneToken, including the gene name

string shortline=strPubMedLine; //declare a new string to store shorten version of PubMedLine

shortline = shortline.erase(strPubMedLine.find(strGeneToken)+strGeneToken.length(), strPubMedLine.length()); //PubMedLine is erased from the end of GeneToken to the end of the line. In other words, string shortline is the PubMedLine upto GeneToken.

for (int i =0; i< countWords(shortline); i++) //Now, for i>0 and i<number of words in shortline, we count the words.

{

tokencount++;

}

/\*........... to here\*/

// Sentence tokenization //

istringstream issPubMedLine(strPubMedLine);

if (f\_Results.is\_open()){

f\_Results << strStandardGeneID<<"\t\t"<<strPMID<<"\t"<<nTotalLineIndex<<'\t'<<'\t'<<countWords(strPubMedLine)<<'\t'<<'\t'<<'\t'<<'\t'<<tokencount<<'\n'; //DATA VALUES for attributes are added//////////////////

}

else cout<<"Unable to open result file!!"<<endl;

}

}

return 0;

}

int geneMatching(ifstream &f\_GeneDic,string strPubMedLine,ofstream &f\_Results,string strPMID)

{

string strGeneDictionaryLine;

while (f\_GeneDic.good()){

string strStandardGeneID;

string strGeneToken;

int nStandardGeneCheker = 0;

// Reading gene dictionary by line //

strGeneDictionaryLine = readLine(f\_GeneDic);

// Dictionary tokenization //

istringstream issGeneName(strGeneDictionaryLine);

geneToeknMatching(issGeneName,strGeneToken,nStandardGeneCheker,strPubMedLine,f\_Results,strPMID);

if (f\_GeneDic.eof() == 1){

f\_GeneDic.clear();

f\_GeneDic.seekg(0,ios::beg);

break;

}

}

return 0;

}

int readAbstract(string strLiteraturefile,string strGeneDicfile,string strOutputfile)

{

// Read dataset //

string strFileName\_PubMedData = strLiteraturefile;

string strFileName\_GeneDictionary = strGeneDicfile;

ifstream f\_PubMedData = readData(strFileName\_PubMedData);

ifstream f\_GeneDic = readData(strFileName\_GeneDictionary);

// Open result data //

ofstream f\_Results (strOutputfile); // ofstream : stream class to write on files

f\_Results<<"GeneName\t\tPMID\tSentenceIndex\tThe#OfTotalTokensInSentence\tLocationOfTokenInSentence\n" ; //ADDED ATTRIBUTES////////////////////

string strPubMedLine;

string strPMID;

while (f\_PubMedData.good()){ // f\_filenaem.good() : check whether state of stream has error or not

// Reading PubMed data by line //

strPubMedLine = readLine(f\_PubMedData);

nTotalLineIndex++;

// PMID Extraction //

strPMID = PMIDExtractor(strPubMedLine,strPMID);

geneMatching(f\_GeneDic,strPubMedLine,f\_Results,strPMID);

}

f\_PubMedData.close();

f\_GeneDic.close();

f\_Results.close();

return 0;

}

/\*

string\* Gene(string Table2)

{

string geneName;

string\* result = new string[256];

int count = 0;

bool already = false;

bool first = true;

ifstream table2(Table2);

if (!table2.is\_open())

cout << "Failed reading data 2" << endl;

else

{

if (first)

{

table2 >> geneName;

first = false;

}

for (int i = 0; i < 5; i++)

table2 >> geneName;

for (int i = 0; i < 256; i++)

{

if (geneName == result[i])

already = true;

}

if (!already)

{

result[count] = geneName;

count++;

}

already = false;

}

return result;

}

\*/

/\*int\* GeneAlone(string\* gene, string Table1, string Table2)

{

bool found = false;

bool first = true;

bool no = false;

string cmp;

int pmid, pmidDis;

int index = 0;

int\* pmidGene = new int[1000];

int\* result = new int[256];

for (int i = 0; i < 256; i++)

result[i] = 0;

ifstream table2(Table2);

if (!table1.is\_open())

cout << "Failed reading data 1" << endl;

if (!table2.is\_open())

cout << "Failed reading data 2" << endl;

else if (table1.is\_open())

{

if (first)

{

table2 >> cmp;

table2 >> cmp;

first = false;

}

while (table2.good())

{

for (int i = 0; i < 4; i++)

table2 >> cmp;

table2 >> pmid;

if (cmp == gene[index])

{

for (int i = 0; i < 1000; i++)

{

if (pmidGene[i] == pmid)

found = true;

}

if (!found)

{

pmidGene[index] = pmid;

index++;

}

found = false;

}

}

ifstream table1(Table1);

int j = 0;

bool forfirst = true;

while (table1.good())

{

if (forfirst)

{

table1 >> pmidDis;

table1 >> pmidDis;

forfirst = false;

}

for (int i = 0; i < 5; i++)

table1 >> pmidDis;

for (int i = 0; i < 1000; i++)

{

if (pmidGene[i] == pmidDis)

pmidGene[i] = 0;

}

if (!no)

{

result[j] = result[j]+1;

}

no = false;

}

for (int i = 0; i < 1000; i++)

{

if (pmidGene[i] != 0)

count++;

}

result[ind] = count;

return result;

}

\*/

/\*

string\*\* chiScore(string Table1, string Table2)

{

ifstream table1(Table1);

if (!table1.is\_open())

cout << "Failed reading data 1" << endl;

int numBoth=0, numDis=0, numGene=0;

if (table1.is\_open())

{

for (int i = 0; i < 5; i++)

{

table1 >> word;

}

while (table1.good())

{

table1 >> DisAtt1;

table1 >> DisAtt2;

table1 >> DisAtt3;

table1 >> DisAtt4;

table1 >> DisAtt5;

ifstream table2(Table2);

if (!table2.is\_open())

cout << "Failed reading data 2" << endl;

else

{

for (int i = 0; i < 5; i++)

{

table2 >> word;

}

while (table2.good())

{

table2 >> GenAtt1;

table2 >> GenAtt2;

table2 >> GenAtt3;

table2 >> GenAtt4;

table2 >> GenAtt5;

if (DisAtt2 == GenAtt2)

{

for (int i = 0; i < 256; i++)

{

if ((GenAtt2 == passed[i]))

{

found = true;

}

}

if (!found)

{

numBoth++;

passed[count] = GenAtt2;

count++;

}

disOnly = false;

}

}

if (disOnly)

numDis++;

}

}

}

}

\*/

string\*\* bestScore(string\*\* table)

{

string \*\*result;

result = new string\* [20];

for(int i = 0; i < 20;i++)

{

result[i] = new string[2];

}

double largest = 0;

int index = 0;

string name = "";

for (int j = 0; j < 20; j ++)

{

largest = 0;

for (int i = 0; i < 256; i++)

{

if (atof(table[i][1].c\_str()) > largest)

{

largest = atof(table[i][1].c\_str());

index = i;

name = table[i][0];

}

}

table[index][1] = "0";

result[j][0] = name;

stringstream ss;

ss << largest;

result[j][1] = ss.str();

}

ofstream f\_Results ("bestScore.txt"); // ofstream : stream class to write on files

f\_Results<<"GeneName\n";

for (int i = 0; i < 20; i++)

{

f\_Results << result[i][0] << endl;

}

cout << "Finished writing." << endl;

return result;

}

string\*\* score(string Table1, string Table2)

{

double repetition = 0;

double newScore = 0;

int Switch = 0;//0 = need new entry

int count = 0;

string word, DisAtt1, DisAtt2;

double DisAtt3, DisAtt4, DisAtt5;

string GenAtt1, GenAtt2;

double GenAtt3, GenAtt4, GenAtt5;

double titleBonus = 1;

string \*\*result;

result = new string\* [256];

for(int index=0; index < 256;index++)

{

result[index] = new string[3];

}

for (int i = 0; i < 256; i++)

{

for (int j = 0; j < 3; j++)

result[i][j] = "";

}

for (int i = 0; i < 256; i++)

{

result[i][2] = "0";

result[i][1] = "0";

result[i][0] = "";

}

double score = 0;

ifstream table1(Table1);

if (!table1.is\_open())

cout << "Failed reading data 1" << endl;

if (table1.is\_open())

{

for (int i = 0; i < 5; i++)

{

table1 >> word;

}

while (table1.good())

{

table1 >> DisAtt1;

table1 >> DisAtt2;

table1 >> DisAtt3;

table1 >> DisAtt4;

table1 >> DisAtt5;

ifstream table2(Table2);

if (!table2.is\_open())

cout << "Failed reading data 2" << endl;

else

{

for (int i = 0; i < 5; i++)

{

table2 >> word;

}

while (table2.good())

{

table2 >> GenAtt1;

table2 >> GenAtt2;

table2 >> GenAtt3;

table2 >> GenAtt4;

table2 >> GenAtt5;

if ((DisAtt2 == GenAtt2) && (DisAtt3 == GenAtt3))

{

if (DisAtt3 == 2)

titleBonus = 10;

//scoring system//

score = (titleBonus + pow(abs(GenAtt4/(DisAtt5 - GenAtt5)), 2));

titleBonus = 1;

//done//

for (int i = 0; i < 256; i++)

{

if ((GenAtt1 == result[i][0]) && (Switch == 0))

{

stringstream ss;

score += atof(result[i][1].c\_str());

ss << score;

result[i][1] = ss.str();

Switch = 1;

repetition = (atof(result[i][2].c\_str())+1);

stringstream sc;

sc << repetition;

result[i][2] = sc.str();

}

}

if (Switch == 0)//need new entry

{

stringstream ss;

result[count][0] = GenAtt1;

ss << score;

result[count][1] = ss.str();

result[count][2] = 1;

repetition = (atof(result[count][2].c\_str())+1);

stringstream sc;

sc << repetition;

result[count][2] = sc.str();

count++;

}

Switch = 0;

score = 0;

}

}

}

}

for (int i = 0; i < 256; i++)

{

newScore = atof(result[i][1].c\_str()) \* (atof(result[i][2].c\_str()));

stringstream ss;

ss << newScore;

result[i][1] = ss.str();

}

}

return result;

}

int main ()

{

string DiseaseTable = "Pancreatic\_cancer\_Disease\_Tagging\_table.txt";

string GeneTable = "Pancreatic\_cancer\_Gene\_Tagging\_table.txt";

bestScore(score(DiseaseTable, GeneTable));

}

Brief Explanation:

|  |  |
| --- | --- |
| Function Name | Role |
| string\*\* score(string Table1, string Table2) | Takes two parameters *Table1* and *Table2*, and access text files named after them. Within each loop, each line from each text file is separated into five different segments of information, and each segment of information from *Table1* is compared to that of *Table2*. Each time they have the same PMID and Sentence Index, the corresponding gene is scored for how closely related it is to the corresponding disease. The function then checks if the relationship between this gene and the disease is scored already. If the gene name already exists in the string matrix *result*, the function adds the new score to the existing score; otherwise, the function simply records the new score as it is in the matrix. After the function is done calculating all the scores, it returns *result*. |
| string\*\* bestScore(string\*\* table) | Takes a parameter called *table*, which is expected to be the return value of the previous function *score*(). This function creates a new string matrix of size *n* and stores *n* genes with the highest scores from *table*. The number *n* can be changed at any time in case the user likes to see how the f-score of the result changes. |

1. Explanation of attributes

|  |  |
| --- | --- |
| Attribute Name | Role |
| string Table1 | Table1 is a parameter. This string stores the name of a text file the function *score*() will read. |
| string Table2 | Table2 is a parameter. This string stores the name of a text file the function *score*() will read. |
| double repetition = 0; | This attribute stores the number of times a gene name appears in the whole data. It is meant to be used as a way to improve the f-score. A repetition value of each gene is saved at the third column of *result*. |
| double newScore = 0; | After the *while* loop inside *score*() is completed, the function will multiply the original scores by *repetition*, obtaining the newScores. These values are then re-saved at the second column of *result*. |
| int Switch = 0;// | Switch will determine when the function needs to add a new entry to *result*. Whenever a new gene name appears as the function goes through the *while* loop, *Switch* will be turned on (*Switch* = 0), and *Switch* will be turned off (*Switch* = 1) when previously stored gene names appear. |
| int count = 0; | This integer value *count* will count the number of genes the function has stored in *result* so far. Whenever the function adds a new entry to *result*, it uses count as the index of the new entry. |
| string word, DisAtt1, DisAtt2 | *word* is used to get rid of unnecessary text segments as the function *score*() reads through text files.  *DisAtt1* saves the first word of a line, which is the name of the disease.  *DisAtt2* saves the second word of a line, which is the PMID. |
| double DisAtt3, DisAtt4, DisAtt5; | *DisAtt3* saves the third word of a line, which is the Sentence Index.  *DisAtt4* saves the fourth word of a line, which is the total number of tokens in the corresponding line.  *DisAtt5* saves the last word of a line, which is the token location of the corresponding disease name. |
| string GenAtt1, GenAtt2; | *GenAtt1* saves the first word of a line, which is the name of the gene.  *GenAtt2* saves the second word of a line, which is the PMID. |
| double GenAtt3, GenAtt4, GenAtt5 | *GenAtt3* saves the third word of a line, which is the Sentence Index.  *GenAtt4* saves the fourth word of a line, which is the total number of tokens in the corresponding line.  *GenAtt5* saves the last word of a line, which is the token location of the corresponding gene name. |
| double titleBonus = 1 | This double value is 1 at default. This value is a constant added to the score in the scoring function. When it is confirmed that both the gene name and disease name appear in the same title, titleBonus is reset to 10, thereby increasing the score by far. |
| string \*\*result | *result* is a 256 x 3 matrix. For a total of 256 genes, the first column saves the gene names, the second column saves their scores, and the last column temporarily saves their numbers of appearances (*repetition*). This matrix is the output (return value) of the function *score*(). |
| double score = 0; | This value temporarily saves the calculated score of a gene in the form of a double. This value will soon be altered to a string form and saved in *result*. |

1. Scoring function with explanation

In my team’s first attempt to create a scoring function, we looked at what we had written in our pre-lab reports. There were the number of co-occurrences in title, the number of co-occurrences in abstracts, the number of times an association word appeared between two terms, the number of times a stop word appeared between two terms, the distance between the two terms, etc. Among them, the variables we could *actually* employ were (a) the number of co-occurrences in title, (b) the number of co-occurrences in abstracts, and (c) the distance between the two terms.

Using those 3 variables only, we set up our first scoring function:

(1)

Unfortunately, function (1) resulted in an f-score of 0.104 with lung cancer (10 best scores). Precision was quite high (around 20%) yet recall was extremely low (around 7 %). To increase recall, we searched for something with the least influence (so that we could be sure that we’d considered all the possibilities, however insignificant and imprecise they may be). Those variables included (d) the rate of the total token to the distance and (e) the number of co-occurrences.

(2)

Still, the f-score increased only by 0.09, resulting in an f-score of 0.111 (10 best scores). Precision and recall values did not change much, so we tried changing the exponent from 1.25 to 2, giving sentence length greater value.

(3)

Neither precision nor recall changed. Indeed, the f-score stayed constant. The size of the exponent did not really matter. So this time, we increased the number of best scores we’re inputting from 10 to 30. The f-score was decreased to 0.099 this time, and both precision and recall were around 0.1.

(4)

I tried changing the titleBonus from a coefficient to a constant value to be added. (30 best scores). The f-score increased back to around 0.1, yet there was not much change. So, I changed the value of titleBonus from 5 to 10, and surprisingly, the f-score increased to 0.147, with precision around 18% and recall around 12%.

(5)

Then I realized that rather than titleBonus, the number of co-occurrences should be the coefficient to reflect how often the gene and the disease are mentioned together. However, the f-score remained unchanged, so did precision and recall (30 best scores).

In our first attempt at pancreatic cancer test, we got an f-score of 0.133. This time, precision (exactly 0.1) was extremely low compared to recall (exactly 0.2). We tried changing the number of best scores back to 10. Surprisingly, the f-score increased from 0.133 to 0.272 with precision of 0.3 and recall of 0.2. Since recall value did not change at all, we realized that there were no true-positive among the last 20 best scores. Later, it turned out that the first three best scores were all true-positive (a test with only the first three best-scores resulted in the precision of 1)

* In all the attempts, my team removed all unnecessary items such as *cell*, *large*, etc.
* With TA Dongjin Jang’s advice, my team tried creating a scoring function using chi square distribution test. In my effort to obtain the disease names, their corresponding gene names, the number of PMIDs containing the corresponding gene name only, and the number of PMIDs containing both the gene name and the disease name, I wrote some code down–they are now saved as comment in the middle of the source code.