Sung Joon Lim

CS 540

HW6

04.01.2019

**Problem 1.1)**

occurrence Count:170581

word Count:18788

used JAVA to read text files and used Map to map them by key and value. Keys are the unique computer word and values are the number of occurrence of the word.

**1.2)**

the 8651

to 4663

a 3673

in 3521

and 3446

of 2792

for 1771

is 1470

on 1432

was 1421

he 1244

with 1166

have 1152

at 1137

I 1126

his1 111

that 1060

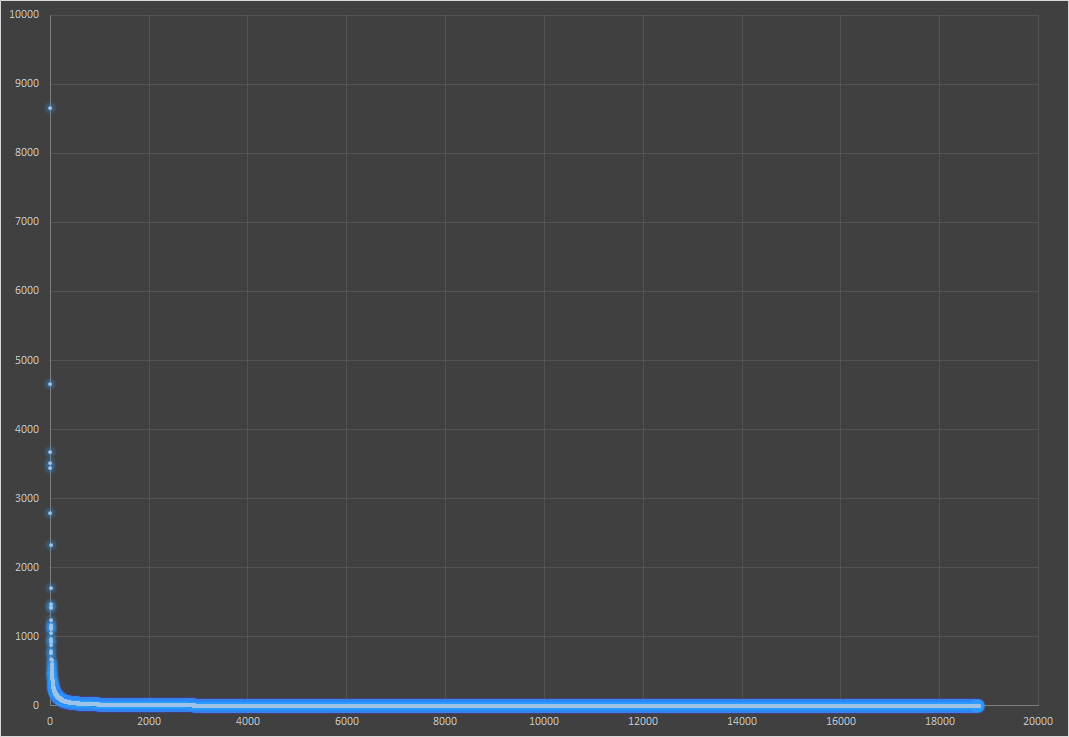
has 965

be 950

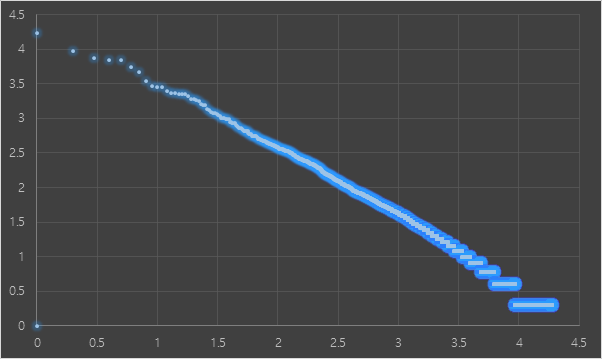
but 931

assigned in array list in order of values

**1.3)**



The raw graph shows that value decrease as rank decrease. It shows that first several words are repeated a lot. However, it’s hard to tell since most of keys are close to zero range and scale for y axes is huge.



The log graph also shows that value decrease as rank decrease. However, the scale for axes are smaller making it better to figure out the trend. It is clear on log graph to figure out that occurrences decrease as ranks decrease.

Used excel to create plot for data.

**1.4)**

4.

tfIdf of word contract = 0.54731

top 10 words:

Ronaldo

contract

United

World.

tomorrow.

first-team.

Trafford.

five-year-deal,

knows,”

club.

knows

“Nobody

resolved

agreeing

star,

sides.”

renew

future.”

News

Portugal

**1.5)**

The cosine similarity of v1 and v2 using bag of words: 0.99885

tfIdf: 0.98863

The values of cosine similarity are different. They are different because bag of words uses word count while tfIdf counts the significance of key.

1.6)

One issue of computer word is that it counts words like conjunctions that appears a lot but are meaningless and insignificant.

Another issue is that it’s not case sensitive. Moreover, special characters like commas and quotations attached can make the same word look different.

**Problem 2.1)**

11 dimensions

mean value of Retail: 32511.33146

mean value of horsepower: 213.2191011

Used =AVERAGE() and =STDEV in excel.

**2.2)**

First second third

|  |  |  |  |
| --- | --- | --- | --- |
| eigenvalue | 7.167308 | 1.826552 | 0.834489 |
| Retail($) | 0.275262 | 0.444142 | 0.259044 |
| Dealer($) | 0.273531 | 0.44631 | 0.261497 |
| Engine(L) | 0.345189 | -0.0135 | 0.064369 |
| Cylinders | 0.332857 | 0.092837 | 0.116059 |
| Horsepower | 0.318994 | 0.280542 | 0.094543 |
| CityMPG | -0.30787 | -0.01645 | 0.547298 |
| HighwayMPG | -0.30506 | -0.03009 | 0.605168 |
| Weight(lb) | 0.335202 | -0.1593 | -0.11482 |
| Wheelbase(in) | 0.263903 | -0.43141 | 0.241694 |
| Length(in) | 0.250374 | -0.43847 | 0.312575 |
| Width(in) | 0.291832 | -0.33334 | 0.05384 |

Used initial data, average value, and standard deviation to get centered and normalized data using given formula. After getting covariance matrix, used excel visual basic to get eigenvectors and eigenvalue.

**2.3)**

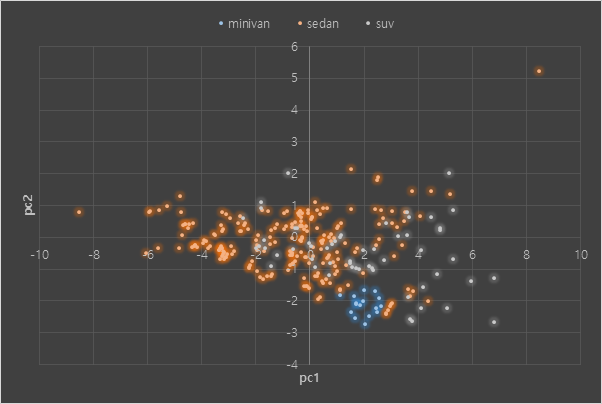
First igenvector has coordinate.

|  |  |
| --- | --- |
| Retail($) | 0.275262 |
| Dealer($) | 0.273531 |
| Engine(L) | 0.345189 |
| Cylinders | 0.332857 |
| Horsepower | 0.318994 |
| CityMPG | -0.30787 |
| HighwayMPG | -0.30506 |
| Weight(lb) | 0.335202 |
| Wheelbase(in) | 0.263903 |
| Length(in) | 0.250374 |
| Width(in) | 0.291832 |

Every features except for CityMPG and HighwayMPG had positive correlation.

Since igenvectors show components direction or slope, positive coordinates show how important each principal is and that values had pattern of positive correlation. Higher igenvalue shows that there’s large variation in that dimension and how important and dominant the component is.

**2.4)**



Used normalized data of minivan, sedan, SUV to create covariance matrix and used =MMULT() command to perform matrix multiplication with first two igenvectors. Made graph with the output.

**2.5)**

Based on the plot, it seems that values for minivans are clustered most strongly. Since igenvalues of pc1 and pc2 were strongly dominant, it is plausible to believe that plot successfully represents original data. However, pc1 is much more important than pc2. Data for sedan and SUV seems to be spread out over pc1 while data for minivan are clustered. So, data for minivan are highly correlated to each other. Since data of minivan are similar to each other, they are likely to be the reasonable merchandise. So, Bob can choose to buy a minivan to avoid making poor choices. Since data for sedan and SUV are not similar to each other, Bob should make careful decisions. However, there are multiple outliers that might result from car brands and other factors.

**Appendix**

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.io.PrintStream;

**import** java.util.\*;

**public** **class** NLP {

Map<String, Double> map = **new** HashMap<>();

// construct dictionary with key and value

**public** **void** makeDict(File file) **throws** FileNotFoundException{

Scanner scr = **new** Scanner(file);

**while** (scr.hasNextLine()){

String[] line = scr.nextLine().split(" ");

**for** (**int** i = 0; i < line.length; i++){

String s = line[i];

**if** (!map.containsKey(s)){

map.put(s, 1.0);

}

**else**{

map.put(s, map.get(s) + 1);

}

}

scr.close();

}

}

// count the number of words in a document

**public** **int** countWords(Map<String, Double> map){

**int** count = 0;

**for** (**int** i = 0; i < map.size(); i++){

count++;

}

**return** count;

}

// number of docs of certain word

**public** **int** getDocCount(String word, File[] files) **throws** FileNotFoundException{

**int** docs = 0;

**boolean** isdoc;

**for** (**int** i = 0; i < files.length; i++){

isdoc = **false**;

**if** (files[i].isFile()){

isdoc = isDoc(word, files[i]);

}

**if** (isdoc == **true**){

docs++;

}

}

**return** docs;

}

//tf-idf

**public** Map<String,Double> ifIdf(HashMap<String,Double> map, File[] files, Map<String, Double> allMap) **throws** FileNotFoundException{

**double** max = 0;

**for** (String keys : map.keySet()){

max = map.get(keys);

**break**;

}

**for** (String keys : map.keySet()){

**double** tf = (**double**)map.get(keys) / max;

**double** docCount = getDocCount(keys, files);

**double** idf = Math.*log10*(511/docCount);

**double** tfidf = tf \* idf;

allMap.put(keys, tfidf);

}

**for** (String keys : allMap.keySet()){

**if** (!map.containsKey(keys)){

allMap.put(keys, 0.0);

}

}

**return** allMap;

}

// bagOfWords

**public** Map<String,Double> bagOfWords(HashMap<String,Double> map, Map<String, Double> allMap) **throws** FileNotFoundException{

**double** sum = 0;

**for** (String keys : allMap.keySet()){

**if** (!map.containsKey(keys)){

allMap.put(keys, 0.0);

}

}

**for** (String keys : map.keySet()){

sum += map.get(keys);

}

**for** (String keys : allMap.keySet()){

allMap.put(keys, allMap.get(keys) / sum);

}

**return** allMap;

}

// returns true if a word is present in the document

**public** **boolean** isDoc(String word, File file) **throws** FileNotFoundException{

**boolean** isdoc = **false**;

Scanner scr = **new** Scanner(file);

**while** (scr.hasNextLine()){

String[]line = scr.nextLine().split(" ");

**for** (**int** i = 0; i < line.length; i++)

{

String s = line[i];

**if** (s.equals(word))

{

isdoc = **true**;

**break**;

}

}

}

scr.close();

**return** isdoc;

}

// hashmap

**public** HashMap<String, Double> sortMap(Map<String, Double> map){

List<Map.Entry<String, Double> > list = **new** LinkedList<Map.Entry<String, Double> >(map.entrySet());

Collections.*sort*(list, **new** Comparator<Map.Entry<String, Double> >() {

**public** **int** compare(Map.Entry<String, Double> o1,

Map.Entry<String, Double> o2) {

**return** (o2.getValue()).compareTo(o1.getValue());

}

});

HashMap<String, Double> curr = **new** LinkedHashMap<String, Double>();

**for** (Map.Entry<String, Double> aa : list) {

curr.put(aa.getKey(), aa.getValue());

}

**return** curr;

}

// sort the hashmap

**public** HashMap<String, Double> sortMapDouble(Map<String, Double> map){

List<Map.Entry<String, Double> > list = **new** LinkedList<Map.Entry<String, Double> >(map.entrySet());

Collections.*sort*(list, **new** Comparator<Map.Entry<String, Double> >() {

**public** **int** compare(Map.Entry<String, Double> o1,

Map.Entry<String, Double> o2) {

**return** (o2.getValue()).compareTo(o1.getValue());

}

});

HashMap<String, Double> curr = **new** LinkedHashMap<String, Double>();

**for** (Map.Entry<String, Double> aa : list) {

curr.put(aa.getKey(), aa.getValue());

}

**return** curr;

}

// cosine similarity

**public** **static** **double** cosineSimilarity(Map<String, Double> map, Map<String, Double> mapB){

**double** sum = 0.0;

**double** normA = 0.0;

**double** normB = 0.0;

**for** (**int** i = 0; i < map.size(); i++){

Object keyA = map.keySet().toArray()[i];

Object keyB = mapB.keySet().toArray()[i];

**double** valA = map.get(keyA);

**double** valB = mapB.get(keyB);

sum += valA \* valB;

normA += Math.*pow*(map.get(keyA), 2);

normB += Math.*pow*(mapB.get(keyB), 2);

}

**return** sum / (Math.*sqrt*(normA) \* Math.*sqrt*(normB));

}

**public** **static** **void** main(String args[]) **throws** FileNotFoundException{

**int** count = 0;

File folder = **new** File("/Users/user/Desktop/CS 540/projects/hw6/src/news");

File[] listOfFile = folder.listFiles();

NLP a = **new** NLP();

**for** (**int** i = 0; i < listOfFile.length; i++) {

a.makeDict(listOfFile[i]);

}

HashMap<String, Double> sorted = **new** HashMap<>();

sorted = a.sortMap(a.map);

**for** (String keys : sorted.keySet()){

**if** (count < 21){

System.***out***.println(keys + " " + sorted.get(keys));

count++;

}

}

}

}