CIND-110
Data Organization for Data Analysts

Lab Manual Module 9
Information Retrieval

Lead Instructor: Dr. Tamer ABDOU

Contents

Objectives

To demonstrate Vector Space Model of Information Retrieval in R by building a search engine.

1. Installation

This section is required only if latest R version is not installed.
From your Ubuntu Terminal, run the commands below.

Installing R

```
1 sudo apt-key adv --keyserver keyserver.ubuntu.com
    --recv-keys
  E298A3A825C0D65DFD57CBB651716619E084DAB9
2 sudo add-apt-repository 'deb [arch=amd64,i386]
  https://cran.rstudio.com/bin/linux/ubuntu
  xenial/'
3 sudo apt-get update
4 sudo apt-get install r-base
5 sudo -i R https://www.digitalocean.com/community/
  tutorials/how-to-install-r-on-ubuntu-16-04-2
```

Then install RStudio through following link: https://www.rstudio.com/products/rstudio/download/. Scroll down and download RStudio 1.0.153-Ubuntu 16.04+/Debian 9+ (64-bit).

RStudio Desktop 1.0.153 — Release Notes

RStudio requires R 2.11.1+. If you don't already have R, download it here.

Installers for Supported Platforms

Installers	Size	Date	MD5
RStudio 1.0.153 - Windows Vista/7/8/10	81.9 MB	2017-07-20	b3b4bbc82865ab105c21cb70b17271b3
RStudio 1.0.153 - Mac OS X 10.6+ (64-bit)	71.2 MB	2017-07-20	8773610566b74ec3e1a88b2fdb10c8b5
RStudio 1.0.153 - Ubuntu 12.04-15.10/Debian 8 (32-bit)	85.5 MB	2017-07-20	981be44f91fc07e5f69f52330da32659
RStudio 1.0.153 - Ubuntu 12.04-15.10/Debian 8 (64-bit)	91.7 MB	2017-07-20	2d0769bea2bf6041511d6901a1cf69c3
RStudio 1.0.153 - Ubuntu 16.04+/Debian 9+ (64-bit)	61.9 MB	2017-07-20	d584cbab01041777a15d62cbef69a976
RStudio 1.0.153 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)	84.7 MB	2017-07-20	8dfee96059b05a063c49b705eca0ceb4
RStudio 1.0.153 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)	85.7 MB	2017-07-20	16c2c8334f961c65d9bfa8fb813ad7e7

Now you are good to go. Follow the manual and execute the commands in RStudio.

2. Creating the Documents

Information retrieval is the process of retrieving documents from a collection in response to a query (or a search request) by a user.
In this section, we explore our documents and the search query.
Note that all queries below must be run in R

Collection of documents

```
1  doc1 <- "Stray cats are running all over the
       place. I see 10 a day!"
2  doc2 <- "Cats are killers. They kill billions of
       animals a year."
3  doc3 <- "The best food in Columbus, OH is   the
       North Market."
4  doc4 <- "Brand A is the best tasting cat food
       around. Your cat will love it."
5  doc5 <- "Buy Brand C cat food for your cat. Brand
        C makes healthy and happy cats."
6  doc6 <- "The Arnold Classic came to town this
       weekend. It reminds us to be healthy."
7  doc7 <- "I have nothing to say. In summary, I
       have told you nothing."
8  doc.list <- list(doc1, doc2, doc3, doc4, doc5,
       doc6, doc7)
9  N.docs <- length(doc.list)
10 names(doc.list) <- paste0("doc",c(1:N.docs))
```

We also have an information need that is expressed via the following text query

```
1  query <- "Healthy cat food"
```

In order to meet our information need amidst all this unstructured text, we are going to implement the vector space model of information retrieval in R. In the process, we will learn something about the TM package and about the analysis of unstructured data before it was Big.

3. Text Mining

Install the TM and SNOWBALL packages on RStudio.

Installing Packages

```
install.packages("slam")
install.packages("SnowballC")
install.packages("tm")
```

Loading the packages into memory

```
library(tm)
library(SnowballC)
library(slam)
```

In text mining and related fields, a corpus is a collection of texts, often with extensive manual annotation. Not surprisingly, the CORPUS class is a fundamental data structure in TM.

Corpus

```
my.docs <- VectorSource(c(doc.list, query))
my.docs$Names <- c(names(doc.list), "query")
my.corpus <- Corpus(my.docs)
my.corpus
```

```
> my.corpus
<<SimpleCorpus>>
Metadata:  corpus specific: 1, document level (indexed): 0
Content:  documents: 8
```

Above, we treated the query like any other document. It is, after all, just another string of text. Queries are not typically known a priori, but in the processing steps that follow, we will pretend like we knew ours in advance to avoid repeating steps.

Standardizing. One of the nice things about the Corpus class is the TM_MAP function, which cleans and standardizes documents within a Corpus object. Below are some of the transformations.

```
1 getTransformations()
```

```
> getTransformations()
[1] "removeNumbers"     "removePunctuation" "removeWords"       "stemDocument"
[5] "stripWhitespace"
```

First, let's remove the punctuation.

Removing Punctuation

```
1 my.corpus <- tm_map(my.corpus, removePunctuation)
2 content(my.corpus[[1]])
```

```
> content(my.corpus[[1]])
[1] "Stray cats are running all over the place I see 10 a day"
```

Suppose we don't want to count "cats" and "cat" as two separate words. To implement the famous Porter Stemmer algorithm. To use this particular transformation, first load the SNOWBALL package.

Stemming

```
1 library(SnowballC)
2 my.corpus <- tm_map(my.corpus, stemDocument)
3 content(my.corpus[[1]])
```

```
> content(my.corpus[[1]])
[1] "Stray cat are run all over the place I see 10 a day"
```

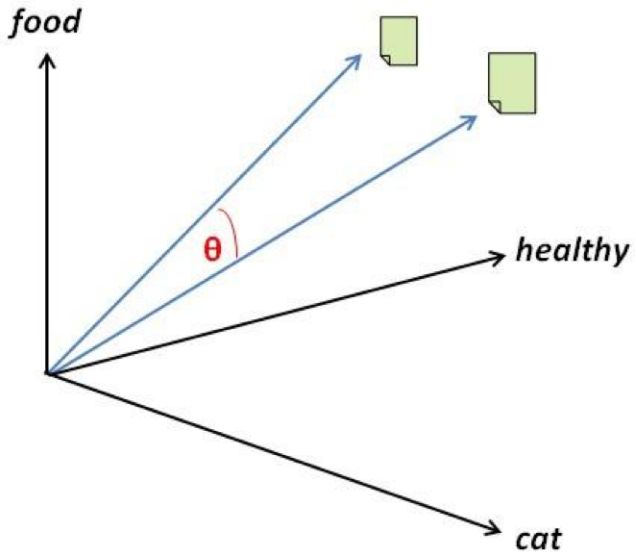
Finally, let's remove numbers and any extra white space/

Remove Numbers and Strip White Space

```
1 my.corpus <- tm_map(my.corpus, removeNumbers)
2 my.corpus <- tm_map(my.corpus,
     content_transformer(tolower))
3 my.corpus <- tm_map(my.corpus, stripWhitespace)
4 content(my.corpus[[1]])
```

```
> content(my.corpus[[1]])
[1] "stray cat are run all over the place i see a day"
```

4. The Vector Space Model



Document Similarity. Here's a trick that's been around for a while: represent each document as a vector in R^N with N as the number of words and we use the angle θ between the vectors as a similarity measure. Rank by the similarity of each document to the query and we get a search engine.
One of the simplest things we can do is to count words within documents. This naturally forms a two dimensional structure, the term document matrix, with rows corresponding to the words and the columns corresponding to the documents. As with any matrix, we may think of a term document matrix as a collection of column vectors existing in a space defined by the rows. The query lives in this space as well, though in practice we wouldn't know it beforehand.

R Commands

```
term.doc.matrix.stm <- TermDocumentMatrix(my.
    corpus); colnames(term.doc.matrix.stm) <- c(
    names(doc.list), "query"); inspect(term.doc.
    matrix.stm[0:14, ])
```

```
> inspect(term.doc.matrix.stm[0:14, ])
<<TermDocumentMatrix (terms: 14, documents: 8)>>
Non-/sparse entries: 22/90
Sparsity           : 80%
Maximal term length: 7
Weighting          : term frequency (tf)
Sample             :
       Docs
Terms   1 2 3 4 5 6 7 8
  all   1 0 0 0 0 0 0 0
  are   1 1 0 0 0 0 0 0
  cat   1 1 0 2 3 0 0 1
  day   1 0 0 0 0 0 0 0
  over  1 0 0 0 0 0 0 0
  place 1 0 0 0 0 0 0 0
  run   1 0 0 0 0 0 0 0
  see   1 0 0 0 0 0 0 0
  stray 1 0 0 0 0 0 0 0
  the   1 0 2 1 0 1 0 0
```

Sparsity and storage of the term document matrix

The matrices in *tm* are of type *Simple Triplet Matrix* where only the triples (i,j,value) are stored for non-zero values. To work directly with these objects, you may use install the slam package. We bear some extra cost by making the matrix "dense" (i.e., storing all the zeros) below.

R Commands

```
term.doc.matrix <- as.matrix(term.doc.matrix.stm)
cat("Dense matrix representation costs", object.size(term.doc.matrix), "bytes.$\n$",      "
Simple triplet matrix representation costs",
object.size(term.doc.matrix.stm),      "bytes."
)
```

```
> term.doc.matrix <- as.matrix(term.doc.matrix.stm)
> cat("Dense matrix representation costs", object.size(term.doc.matrix), "bytes.\n",
+     "Simple triplet matrix representation costs", object.size(term.doc.matrix.stm),
+     "bytes.")
Dense matrix representation costs 6688 bytes.
 Simple triplet matrix representation costs 5808 bytes.
```

Implementation

For both the document and query, we choose tfidf weights of (1 + log2(tf)) *log2(N/ tf). Note that whenever a term does not occur in a specific document, or when it appears in every document, its weight is zero.

Function computing tfidf weights from term frequency vector

```
get.tf.idf.weights <- function(tf.vec) { n.docs
<- length(tf.vec);   doc.frequency <- length(tf
.vec[tf.vec > 0]);   weights <- rep(0, length(
tf.vec));   weights[tf.vec > 0] <- (1 + log2(tf
.vec[tf.vec > 0])) * log2(n.docs/doc.frequency)
 return(weights) }
#For a word appearing in 4 of 6 documents,
occurring 1, 2, 3, and 6 times:
get.tf.idf.weights(c(1, 2, 3, 0, 0, 6))
```

```
> get.tf.idf.weights(c(1, 2, 3, 0, 0, 6))
[1] 0.5849625 1.1699250 1.5121061 0.0000000 0.0000000 2.0970686
>
```

Using *apply*, we run the tfidf weighting function on every row of the term document matrix. The document frequency is easily derived from each row by the counting the non-zero entries (not including the query).

Function computing tfidf weights from term frequency vector

```
1 tfidf.matrix <- t(apply(term.doc.matrix, 1, FUN =
      function(row) {get.tf.idf.weights(row)}))
2 colnames(tfidf.matrix) <- colnames(term.doc.
     matrix)
3 tfidf.matrix[0:3, ]
```

```
> tfidf.matrix[0:3, ]

Terms         1         2 3        4       5 6 7         8
  all 3.0000000 0.0000000 0 0.000000 0.00000 0 0 0.0000000
  are 2.0000000 2.0000000 0 0.000000 0.00000 0 0 0.0000000
  cat 0.6780719 0.6780719 0 1.356144 1.75279 0 0 0.6780719
```

Dot Product Geometry

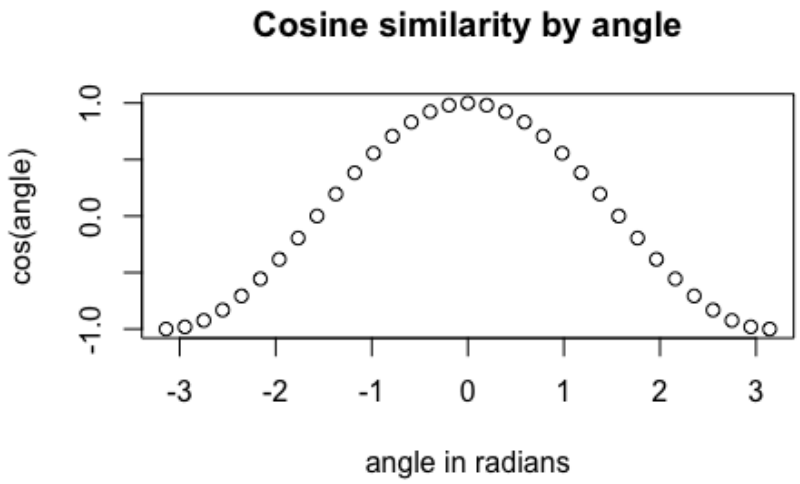
A benefit of being in the vector space is the use of its dot product. For vectors a and b, the geometric definition of the dot product is a cdot = | | a | | * | | b | | *cos theta where | | cdot | | is the Euclidean norm (the root sum of squares) and theta is the angle between a and b.

If a and b are orthogonal, then a.b=0
If they are co-directional, then a.b= | | a | | | | b | | .
The dot product of a vector by itself is, a.a= | | a | | $\hat{2}$ where | | a | | = $\sqrt{a.a}$ which is the Euclidean Length of the vector.

In fact, we can work directly with the cosine of theta. For theta in the interval [-pi ,pi],the endpoints are orthogonality (totally unrelated documents) and the center, zero, is complete collinearity (maximally similar documents). We can see that the cosine decreases from its maximum value of 1 as the angle departs from zero in either direction.

Plot

```
1 angle <- seq(-pi, pi, by = pi/16)
2 plot(cos(angle) ~ angle, type = "b", xlab = "
    angle in radians", main ="Cosine similarity by
    angle")
```



may furthermore normalize each column vector in our tfidf matrix so that its norm is one. Now the dot product is *cos theta*.

R commands

```
1 tfidf.matrix <- scale(tfidf.matrix, center =
    FALSE, scale = sqrt(colSums(tfidf.matrix^2)))
2 tfidf.matrix[0:3,]
```

```
> tfidf.matrix[0:3, ]

Terms         1         2 3         4         5 6 7         8
  all 0.36257971 0.00000000 0 0.0000000 0.0000000 0 0 0.0000000
  are 0.24171981 0.26157811 0 0.0000000 0.0000000 0 0 0.0000000
  cat 0.08195171 0.08868438 0 0.1883549 0.2079088 0 0 0.3644225
```

Dot Product Machine

Keeping the query alongside the other documents let us avoid repeating the same steps. But now it's time to pretend it was never there.

R commands

```
query.vector <- tfidf.matrix[, (N.docs + 1)]
tfidf.matrix <- tfidf.matrix[, 1:N.docs]
```

With the query vector and the set of document vectors in hand, it is time to go after the cosine similarities. These are simple dot products as our vectors have been normalized to unit length.

Recall that matrix multiplication is really just a sequence of vector dot products. The matrix operation below returns values of $\cos\theta$ for each document vector and the query vector.

R commands

```
doc.scores <- t(query.vector) %*% tfidf.matrix
```

With scores in hand, rank the documents by their cosine similarities with the query vector.

R commands

```
results.df <- data.frame(doc = names(doc.list),
   score = t(doc.scores), text = unlist(doc.list))
results.df <- results.df[order(results.df$score,
   decreasing = TRUE), ]
```

The results - Evaluating the model

R commands

```
options(width = 2000)
print(results.df, row.names = FALSE, right =
   FALSE, digits = 2)
```

```
> print(results.df, row.names = FALSE, right = FALSE, digits = 2)
 doc  score text
 doc5 0.267 Buy Brand C cat food for your cat. Brand C makes healthy and happy cats.
 doc4 0.143 Brand A is the best tasting cat food around. Your cat will love it.
 doc6 0.132 The Arnold Classic came to town this weekend. It reminds us to be healthy.
 doc3 0.090 The best food in Columbus, OH is   the North Market.
 doc2 0.032 Cats are killers. They kill billions of animals a year.
 doc1 0.030 Stray cats are running all over the place. I see 10 a day!
 doc7 0.000 I have nothing to say. In summary, I have told you nothing.
```

The “best” document, at least in an intuitive sense, comes out ahead with a score nearly twice as high as its nearest competitor.

Notice however that this next competitor has nothing to do with cats. This is due to the relative rareness of the word “healthy” in the documents and our choice to incorporate the inverse document frequency weighting for both documents and query.

We can calculate the precision, recall and F-Score to find the accuracy.

The entire R Script for the above exercise is uploaded on the D2L shell.