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

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

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

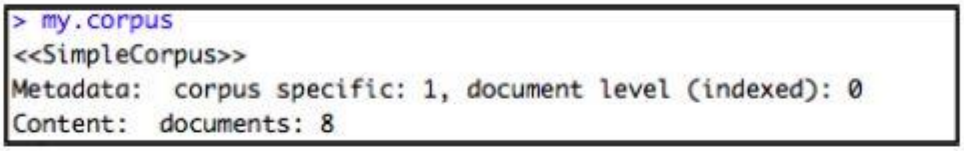
```
query <- "Healthy cat food"
```

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.

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
```

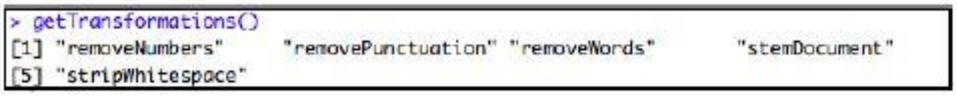


```
> my.corpus
<<SimpleCorpus>>
Metadata:  corpus specific: 1, document level (indexed): 0
Content:  documents: 8
```

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.

```
getTransformations()
```

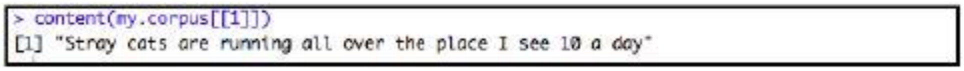


```
> getTransformations()
[1] "removeNumbers"     "removePunctuation" "removeWords"       "stemDocument"
[5] "stripWhitespace"
```

First, let's remove the punctuation.

Removing Punctuation

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

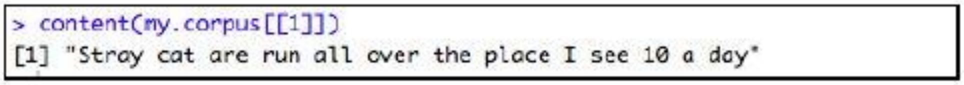


```
> content(my.corpus[[1]])
[1] "Stray cats are running all over the place I see 10 a day"
```

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

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



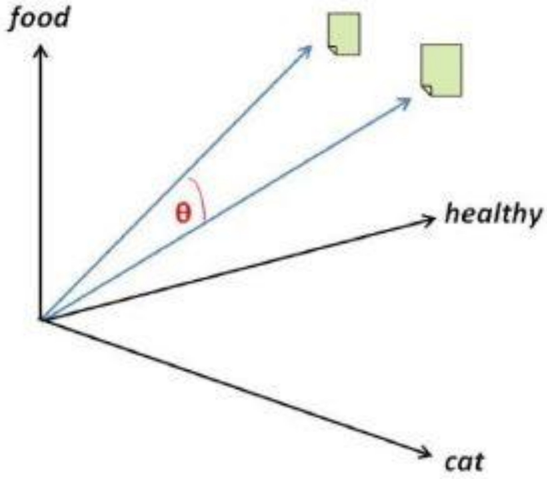
```
> content(my.corpus[[1]])
[1] "Stray cat are run all over the place I see 10 a day"
```

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

Remove Numbers and Strip White Space

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

4. The Vector Space Model

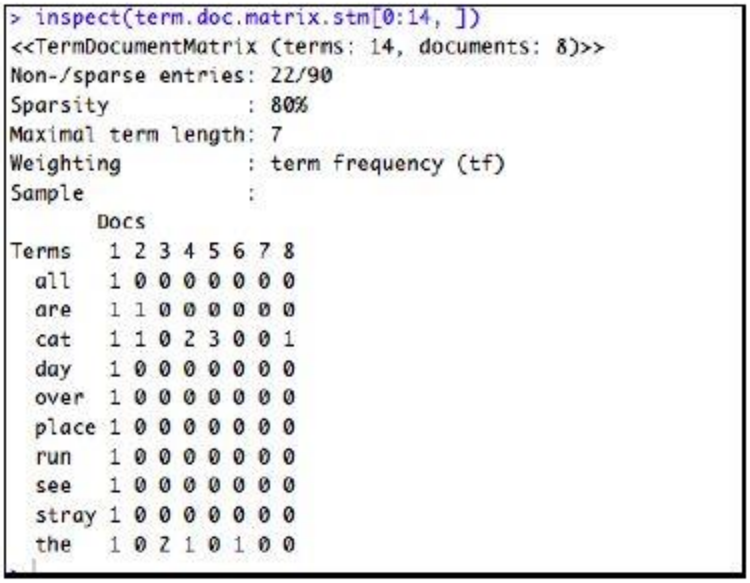


Document Similarity. Here's a trick that's been around for a while: represent each document as a vector in R^N with N as the number of words and we use the angle θ between the vectors as a similarity measure. Rank by the similarity of each document to the query and we get a search engine.

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

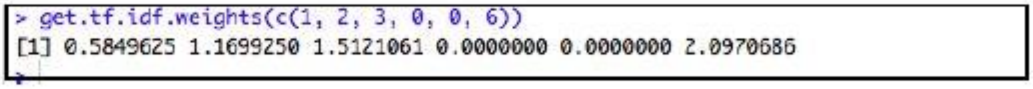
The matrices in *tm* are of type *Simple Triplet Matrix* where only the triples (i,j,value) are stored for non-zero values. To work directly with these objects, you may use install the slam package. We bear some extra cost by making the matrix "dense" (i.e., storing all the zeros) below.

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))
```

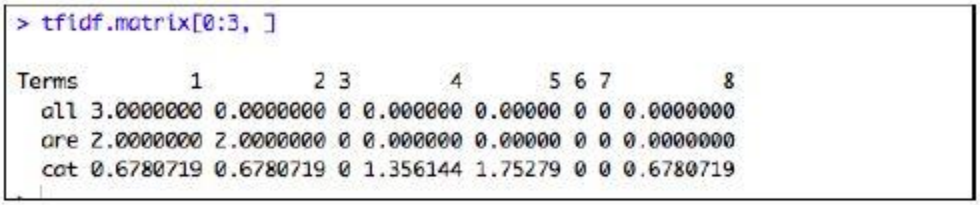


```
> get.tf.idf.weights(c(1, 2, 3, 0, 0, 6))
[1] 0.5849625 1.1699250 1.5121061 0.0000000 0.0000000 2.0970686
>
```

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

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

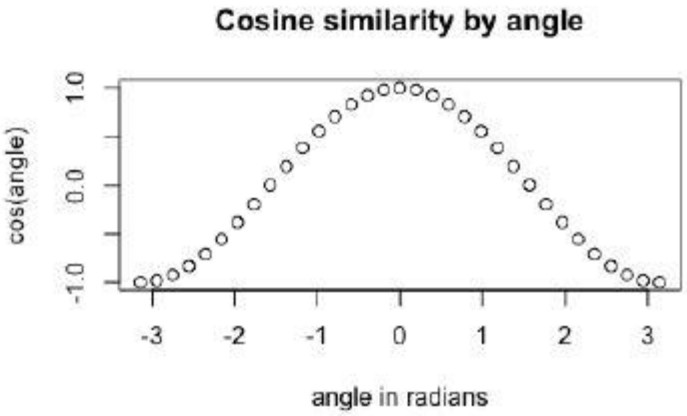
A benefit of being in the vector space is the use of its dot product. For vectors a and b, the geometric definition of the dot product is a cdot = ||a||*||b||*cos theta where ||cdot|| is the Euclidean norm (the root sum of squares) and theta is the angle between a and b.

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||2̂ 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

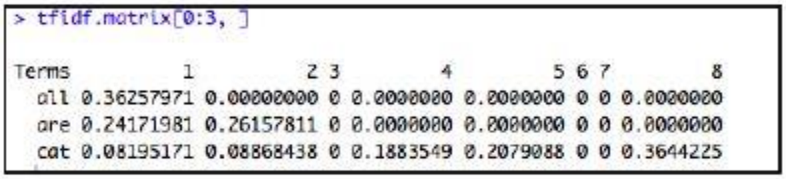
```r
angle <- seq(-pi, pi, by = pi/16)
plot(cos(angle) ~ angle, type = "b", xlab = "
    angle in radians", main ="Cosine similarity by
    angle")
```



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

```r
tfidf.matrix <- scale(tfidf.matrix, center =
    FALSE, scale = sqrt(colSums(tfidf.matrix^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)
```

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.