Homework 1

36-350: Data Mining

SOLUTIONS

1. (a) *What is the bag-of-words representation of the sentence "to be or not to be"?*

   ANSWER: A vector with one component for each word in our dictionary, all of them zero except for the following:

   ```
   be not  or  to
    2   1   1   2
   ```

   This is the format given by

   ```
   table(c("to","be","or","not","to","be"))
   ```

   (b) *Suppose we search for the above sentence via the keyword "be". What is the bag-of-words representation for this query, and what is the Euclidean distance from the sentence?*

   ANSWER: A vector whose only non-zero component is that for "be", where the count is 1. The Euclidean distance is

   $$\sqrt{(2-1)^2 + (1-0)^2 + (1-0)^+(0-2)^2} = \sqrt{7}$$

   (c) *Describe how weighting words by inverse-document-frequency (IDF) should help when making a Web query for "The Principles of Data Mining."*

   ANSWER: It keeps us from wasting time on words like "the" and "of", and emphasize the less-common, more-informative words "principles", "data" and "mining"; something titled "Data Mining Principles" is a good match.

   (d) *Describe a simple text search that could not be carried out effectively using a bag-of-words representation (no matter what distance measure is used). "Simple" means no high-level understanding of English is required.*

   ANSWER: There are many; but a search for the *exact phrase* "to be or not to be" is impossible.

2. (a) *What is the Euclidean distance between each of the vectors* $(1, 0, 0)$*,* $(1, 4, 5)$*, and* $(10, 0, 0)$*?*

   ANSWER: The distance between the first and second vectors is $\sqrt{41} \approx 6.4$; between the first and third is 9; and between the second and third is $\sqrt{122} \approx 11$.

   (b) *Divide each vector by its sum. How do the relative distances change?*

   ANSWER: The first and third vectors become the closest pair.

   (c) *Divide each vector by its Euclidean length. How do the relative distances change?*

   ANSWER: The first and third vectors again become the closest pair.

   (d) *Suppose we're using the bag-of-words representation for similarity searching with a Euclidean metric. Describe how the previous parts of the question illustrate a potential problem if we do not normalize for document length.*

   ANSWER: Documents with the same distribution of words but different sizes will appear to be very far apart.

   (e) *Consider the conventional searching scheme where the user picks a set of keywords and the system returns all documents containing those keywords. Describe how the previous parts of the question illustrate a potential problem with this type of search.*

   ANSWER: The second vector contains the first keyword, but it's a small part of it, and return it because it's a match will lower the precision.

3. (a) *Create document vectors for each of the posts under* `talk.politics.misc` *and* `talk.religion.misc`*. What command would you use to extract the* 57th *word of post number 176845 in* `talk.politics.misc`*? (If this is working right, the word should be "escaped".) Give a command to count the number of times the word "the" appears in that post. (There are at least two ways to do this. The correct answer is 7.)*

   ANSWER: Make the word vector in question by a command like

   ```
   politics.176845 = read.doc("talk.politics.misc/176845.txt")
   ```

   and then extract the component by

   ```
   politics.176845[57]
   ```

   To count the number of times "the" appears, you can use

   ```
   sum(politics.176845 == "the")
   ```

   or

   ```
   table(politics.176845)["the"]
   ```

   I'm sure there are other ways as well.

   (b) *Give the commands you would use to construct a bag-of-words dataframe from the document vectors for the* `talk.politics.misc` *and* `talk.religion.misc` *posts.*

   ANSWER: Something like this would work.

   ```
   pol.rel.list = list(politics.176845,politics.176846,politics.176851,
   politics.176856,politics.176858,politics.176859,politics.176860,politics.176862,
   religion.82757,religion.82759,religion.82762,religion.82764,religion.82765,
   religion.82775,religion.82777,religion.82778)
   pol.rel.names = c("politics.176845","politics.176846","politics.176851",
   "politics.176856","politics.176858","politics.176859","politics.176860",
   "politics.176862","religion.82757","religion.82759","religion.82762",
   "religion.82764","religion.82765","religion.82775","religion.82777",
   "religion.82778")
   pol.rel = make.BoW.frame(pol.rel.list,row.names=pol.rel.names,
                            remove.singletons=FALSE)
   ```

   Note that the row names are optional, but do help to keep track.

   (c) *Create distance matrices from this data frame for (a) the straight Euclidean distance, (b) the distance with sum-of-entries scaling and (c) the distance with vector-length scaling, and then for all three again with inverse-document-frequency weighting. Give the commands you use.*

   ANSWER:

```
pol.rel.dist = distances(pol.rel)
pol.rel.dist.sum = distances(div.by.sum(pol.rel))
pol.rel.dist.len = distances(div.by.euc.length(pol.rel))
pol.rel.idf = idf.weight(pol.rel)
pol.rel.idf.dist = distances(pol.rel.idf)
pol.rel.idf.dist.sum = distances(div.by.sum(pol.rel.idf))
pol.rel.idf.dist.len = distances(div.by.euc.length(pol.rel.idf))
```

(d) *For each of the six different difference measures, what is the average distance between posts in the same newsgroup and between posts in different newsgroups? (Include the R command you use to compute this — don't do it by hand!)*

ANSWER: First, create an array which gives the class labels. We have 16 documents, the first eight of which are about politics, followed by 8 on religion.

```
doc.class.labels = c("politics","politics","politics","politics","politics",
"politics","politics","politics","religion","religion","religion","religion",
"religion","religion","religion","religion")
```

Now create a matrix which says whether or not two documents belong to different classes.

```
ndocs = length(doc.class.labels)
are.different = (matrix(doc.class.labels,nrow=ndocs,ncol=ndocs,byrow=TRUE) !=
matrix(doc.class.labels,nrow=ndocs,ncol=ndocs))
```

The first `matrix()` command makes a square array where each row repeats the class labels; the second one creates a square array where the columns repeat the class labels. The comparison $!=$ checks whether the matrices are the same term by term, so it gives us back a matrix whose `[i,j]` entry is `TRUE` if documents i and j belong to different classes. Now we can get the average distances between documents in different classes:

```
mean(pol.rel.dist[are.different])
mean(pol.rel.dist.sum[are.different])
mean(pol.rel.dist.len[are.different])
mean(pol.rel.idf.dist[are.different])
mean(pol.rel.idf.dist.sum[are.different])
mean(pol.rel.idf.dist.len[are.different])
```

We also need a matrix which says whether two *distinct* documents belong to the same class.

```
are.same = ! are.different
diag(are.same) = FALSE
```

We then compute the average within-class distance by using `are.same` in place of `are.different`

	without IDF			with IDF		
	plain	sum-normed	length-normed	plain	sum-normed	length-normed
within-class distance	52	0.10	0.96	71	0.13	1.4
between-class distance	55	0.11	0.98	72	0.13	1.4

(e) *Create multidimensional scaling plots for the different distances, and describe what you see. Include the code you used, the plots, and explanations for the code.*

ANSWER: I'll use the basic command `cmdscale()`, which returns a matrix whose columns give the new coordinates. I write a little function to plot the points from the two classes in two different colors.

```
plot.cmdscale <- function(distances, labels,class.colors=c("red","blue"),...) {
  # Should really check that the size of distances = (length of labels)^2
  # and that both arguments make sense
  # What are the different labels?
  label.values = unique(labels)
  num.labels = length(label.values)
  # Re-cycle the colors if need be
  if (length(class.colors) < num.labels) {
    class.colors = rep(class.colors,length.out=num.labels)
  }
  # Make it so that we can access colors by the corresponding labels
  names(class.colors) = label.values
  # Make a vector giving the class colors in order
  cols.vec = class.colors[labels]
  # Get the MDS coordinates
  mds.coords = cmdscale(distances)
  # Plot the points: by using a vector for the color argument, we control the
  # color for each point
  plot(mds.coords,col=cols.vec,...)
  return(invisible(mds.coords))
}
```

4. *Comment the `sq.Euc.dist` function — that is, go over it and explain, in English, what each line does, and how the lines work together to calculate the function.*

   ANSWER: The key observation is that $\|\vec{x}-\vec{y}\|^2 = \|\vec{x}\|^2 - 2\vec{x}\cdot\vec{y} + \|\vec{y}\|^2$ for any vectors $\vec{x}$ and $\vec{y}$. (EXERCISE: show this!) The function uses this to efficiently compute the squared distance, avoiding explicit iteration loops (which are slow in R) and avoid computing the same $\|\vec{x}\|^2$ and $\|\vec{y}\|^2$ terms over and over again.

```
sq.Euc.dist <- function(x,y=x) { # Set default value for second argument
  x <- as.matrix(x) # Convert argument to a matrix to make sure
                    # it can be manipulated as such
  y <- as.matrix(y) # Ditto
  nr=nrow(x) # Count the number of rows of the first argument
             # This is the number of rows in the output matrix
  nc=nrow(y) # Count number of rows in the second argument
             # This is the number of COLUMNS in the output
  x2 <- rowSums(x^2) # Find sum of squares for each x vector
  xsq = matrix(x2,nrow=nr,ncol=nc) # Make a matrix where each COLUMN is a copy
                                   # of x2 --- see help(matrix) for details on
                                   # "recycling" of arguments --- matrix is
                                   # sized for output
  y2 <- rowSums(y^2) # Find sum of squares for each y vector
  # Make a matrix where each ROW is a copy of y2, again sized for output
  ysq = matrix(y2,nrow=nr,ncol=nc,byrow=TRUE)
  # Make a matrix whose [i,j] entry is the dot product of x[i,] and y[j,]
  xy = x %*% t(y) # You should check that this has nr rows and nc columns!
  d = xsq + ysq - 2*xy # Add partial result matrices
    # Remember that for vectors x and y
                     # |x-y|^2 = |x|^2 - 2x*y + |y|^2
    # writing * for the dot product.  So d has as its [i,j] entry the squared
    # norm of x[i,] plus the squared norm of y[j,] minus twice their dot product
  # Make the diagonal EXACTLY zero if the two arguments are the same
  if(identical(x,y)) diag(d) = 0
    # Need to use the identical() function to see if two whole objects are the
    # same --- using "x==y" here would give a MATRIX of Boolean values
  # Distances are >= 0, negative values are presumably from numerical errors in
  # calculating numbers close to zero; fix them
  d[which(d < 0)] = 0
  return(d) # Return the tidied-up matrix of squared distances
}
```

   This is more detailed than it strictly needs to be.

5. (a) *Explain what the "cosine distance" has to do with cosines.*

   ANSWER: From vector algebra, we know that for any vectors $\vec{x}$ and $\vec{y}$,

$$\vec{x} \cdot \vec{y} \equiv \sum_i x_i y_i = \|\vec{x}\| \|\vec{y}\| \cos \theta_{\vec{x}\vec{y}}$$

   where $\theta_{\vec{x}\vec{y}}$ is the cosine of the angle between the vectors. The cosine distance is the dot product divided by the product of the norms, so it's that cosine.

   (b) *Calculate, by hand, the cosine distances between the three vectors in question 2.*

   ANSWER: The cosine distance between the first and the third vector is clearly 1, and between either of them and the second vector is ≈ 0.15.

   (c) *Write a function to calculate the matrix of cosine distances (really, similarities) between all the vectors in a data-frame.* Hint: *you may want to use the* `distances` *function. Check that your function agrees with your answer to the previous part.*

   ANSWER: The `distances` function can take an optional argument which is a function to apply to each pair of vectors from the two matrices. So first let's define functions which compute the dot product and the cosine distance for a pair of vectors.

```
dot.product = function(x,y) {
  return(sum(x*y)) # With vectors, * does component-by-component multiplication
}

cosine.dist.pair = function(x,y) {
 dp = dot.product(x,y) # Pass off work to dot.product()
 norm.x = sqrt(sum(x^2)) # Square each component, sum, square root
 norm.y = sqrt(sum(y^2)) # Ditto
 return(dp/(norm.x*norm.y))
}
```

   Now our real function is easy:

```
cosine.dist.1 = function(x) {
  return(distances(x,fun=cosine.dist.pair))
}
```

   This is not the slickest way to do it, because we calculate the norm of each vector $2n-1$ times. (EXERCISE: Why is it $2n-1$?) This is faster, at least when n is large:

```
cosine.dist.2 = function(x) {
  y = div.by.euc.length(x)
  return(distances(y,fun=dot.product))
}
```

EXERCISE: Why does this work? How does this avoid recomputing the norms of the vectors?

6. *Write a function to find the document which best matches a given query string. You can pick the distance measurement, but you should include inverse document-frequency weighting.*

   ANSWER: The easiest way to do this is to use the `nearest.points` function in the provided code. The one tricky bit is making sure that the query string is turned into a bag-of-words vector using the same dictionary as the documents.

```
# Return the index of the document which best matches a given query string
  # First we convert the query string into a document vector, then into
  # a bag of words, and then we remove terms not in the lexicon of the data
  # frame storing the bag-of-words vectors for the documents
    # Note that these last removed words add the same amount to the (squared)
    # distance between the query and any document's bag of words, so those words
    # do not change which document is closest
# Inputs: query (vector of strings), data frame of bag-of-word vectors
# Presumes: a data frame has been created for bag-of-word vectors, column
#           names being words
# Calls: strip.text(), get.idf.weight(), nearest.points()
# Output: Index of the document, name of the corresponding row in data frame
query.by.similarity <- function(query,BoW.frame) {
  # Prepare the query BoW vector for comparison with the target documents
  query.vec = strip.text(query) # Turn the query into a vector of words
  query.BoW = table(query.vec) # Turn it into a bag of words
  lexicon = colnames(BoW.frame) # PRESUMES: BoW.frame has been made so words are
                                # the column names
  query.vocab = names(query.BoW) # What words appear in the query?
  # Restrict the query to words in the lexicon
  query.lex = query.BoW[intersect(query.vocab,lexicon)]
  # Add zero entries for lexicon words not found in the query
  query.lex[setdiff(lexicon,query.vocab)] = 0
  # query.lex now has all the right entries, but the words are out of order!
    # Why does the following trick work?
  query.lex = query.lex[lexicon]
  # Finally, turn it into a one-row matrix
  q = t(as.matrix(query.lex))

  # Do IDF scaling on the targets AND the query
    # The function idf.weight calculates the weights and re-scales the
    # data-frame but we need the weights to re-scale the query as well so we
    # write our own function get.idf.weights() --- see below
  # Get the weights
  idf = get.idf.weights(BoW.frame)
```

```
  # Scale the columns in BoW.frame
  BoW = scale.cols(BoW.frame,idf)
  # Scale the columns in q
  q = q * idf

  # Scale the rows by Euclidean length
  BoW = div.by.euc.length(BoW)
  # Ditto the query
  q = q/sqrt(sum(q^2))

  # Find the closest match between q and BoW.idf
  best.index = nearest.points(q,BoW)$which
  # Use row names from the data for a little extra comprehensibility
  best.name = rownames(BoW)[best.index]
  return(list(best.index=best.index,best.name=best.name))
}

# Return the vector of IDF weights from a data frame
  # The core of idf.weight(), but leaving out the actual re-scaling
# Input: data frame
# Output: weight vector
get.idf.weights <- function(x) {
  doc.freq <- colSums(x>0)
  doc.freq[doc.freq == 0] <- 1
  w <- log(nrow(x)/doc.freq)
  return(w)
}
```

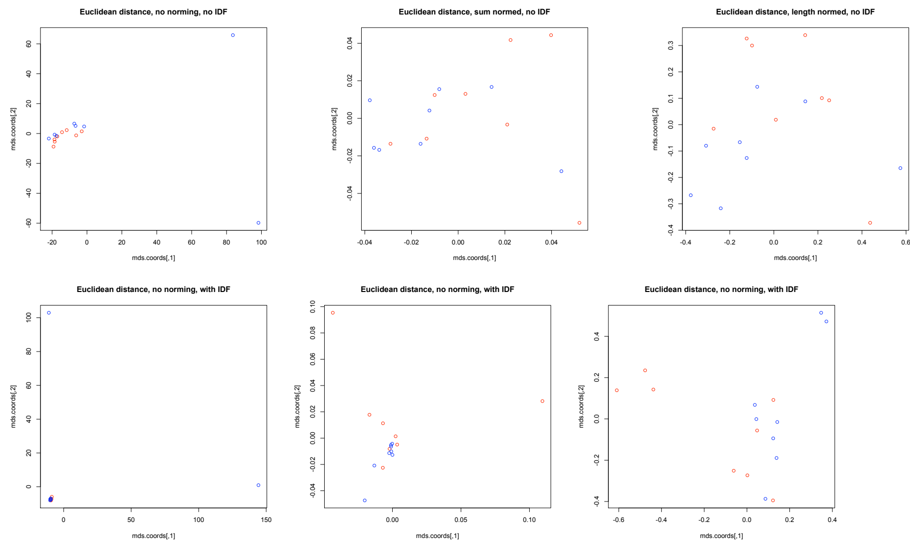

Figure 1: MDS plots. Top row, without IDF. Bottom row, with IDF. Left column, un-normalized vectors. Middle column, normalized by sum. Right column, normalized by Euclidean length. Red indicates politics, blue religion.