0 Review

0.1 Finding indices that satisfy certain conditions

Use of various versions of the `which` command can be useful for determining what parts of a matrix satisfy certain contraints without using a loop. For example suppose the vector `X` contains 50 iid $N(0, 1)$ realizations:

```
X = rnorm(50, mean=0, sd=1)
```

If you want to determine the indices of `X` that are less than or equal to 0 and *scale up* those entries by 10, you can type

```
# find the indices of X that are less than or equal to 0
w <- which(X <= 0)

# scale up X at those indices by 10
X[w] <- 10*X[w]

print(X)
```

If you want to determine what indices of `X` correspond to the max and min of the `X` you can type

```
# this is equivalent to wmin <- which(X==min(X))
wmin <- which.min(X)

# this is equivalent to wmax <- which(X==max(X))
wmax <- which.max(X)
```

Similar operations can be carried out using `which`. For example suppose you want to find the entry of `X` that is closest to its sample mean you could type

```
which.min( abs(X - mean(X)) )
```

Notice how much simpler this is than a loop which does the same thing:

```
# storage for the index closest to the sample mean
m <- Inf; wm <- 0
for(i in 1:length(X))
{

   if( abs(X[i] - mean(X)) < m )
   {

      m <- abs(X[i] - mean(X))
      wm <- i

   }

}
```

0.2 Calculating proportions of a vector that satisfy a certain condition

Keeping the same X defined above, suppose you wanted to calculate the proportion of X that is less than 1– you can do this by defining a variable that is the indicator that X is between 1 and 2, and calculate the mean of this variable:

```
# the indicator that X <= 1
Z <- ( (X <= 1) & (X >= 0) )
Z
mean(Z)
```

Notice how much simpler this is than the following program that does the same thing:

```
# variable the stores the number of elements in X between 0 and 1
count <- 0

for(i in 1:length(X))
{

   if( (X[i] <= 1) & (X[i] >= 0) )
   {

      count <- count + 1

   }

}
count/length(X)
```

See the notes from Lab 1 for several more examples.

0.3 Generality in programming

when you are asked to "write your code in a general way", this means that your code can *only depend on your knowledge of the dataset's structure, not on your inspection of the dataset by eye.* Using the chips dataset from homework 2 as an example, suppose you are asked to scale the values in the Data variable by its median.

```
Y <- read.table('chips.dt', header=T, row.names=1)
Y$Data
```

Clearly the median of this variable is 32, so we could just type

```
## Y$Data <- Y$Data/32
```

to accomplish the desired task. It is simple in this case since the dataset is so small, but this is not a good habit in general, since most datasets are much larger and inspection by eye can not be relied on. The general way to do this scaling would be

```
Y$Data <- Y$Data/median(Y$Data)
```

Along these same lines, it is better to use the `$` to select variables from a data frame, instead of the regular indexing by column number. In this case, it is easy to see by eye that `Data` corresponds to the 6th column, so you would select that variable by typing `Y[,6]`, but some datasets will have many columns so relying on counting columns will not be feasible.

0.4 The `attach` command

Recall the `attach` command is useful when you will be using a dataset and don't want to repeatedly type `$` whenever you access a variable. For example if you typed

```
attach(Y)
```

you can now access the variable `Data` by typing `Data` instead of `Y$Data`. One major issue that must be noted here in that **if you alter a variable without using `$` after attaching the dataset, then the original data frame will remain unaltered**. For example,

```
Data <- rep(0, length(Data))
Data
Y$Data
```

Notice `Y$Data` remains unchanged. The `attach` command simply creates a copy of each column of the data frame and stores the copies into variables that have the same name as the columns– **they are not referencing the same part of R's memory that the columns of your data frame are**. To alter the dataset you still need to use the `$`.

0.5 Simulating an event occurring with probability p

Supposed you are encountered with a problem that tells you to make something happen with a particular probability, call it p. Such problems can be simulated by generating a single bernoulli trial that has success probability p, and if the result is '1', then the event occurred and otherwise it did not. You can generate a bernoulli trial with success probability p one of two ways:

```
p <- .7

# generate a binomial with number of trials = 1
rbinom(1, 1, p)

# generate the indicator that a uniform(0,1) is less than p
(runif(1) < p)
```

The first one is by the definition of the binomial distribution, and the second is because if $U \sim \text{Uniform}(0,1)$, and $I = \mathcal{I}(U \leq p)$, then

$$P(I = 1) = P(U \leq p) = \int_0^p dx = p$$

so the indicator that a Uniform(0,1) is less than p is 1 with probability p. As an example, reall the homework problem from last week you were supposed to make an immune person lose their immunity with probability p_l. This is accomplished by

```
p_l <- .41
person <- "immune"

if( runif(1) < p )  person <- "susceptible"
```

we can check that this indeed makes the person susceptible with probability p_l:

```
people <- rep("immune", 10000)
for(i in 1:10000)
{
 if( runif(1) < p_l ) people[i] <- "susceptible"
}
mean( people == "susceptible" )
```

So making an event occur with a specified probability simply amounts to generating bernoulli trials. A more complicated example is given in the lab 3 notes, on the disease process example.

0.6 Programming style

The only real style I ask you to observe is indentation and proper placement of braces. When this is not obeyed and it is clear that your code does not work, it makes it difficult to find where the error occurred. It will also make it difficult for you to understand what your code did when you look at it later. Here is an of good and bad style:

```
A = matrix(0, 50, 50)

# bad style
for(i in 1:50){
for(j in 1:50){
if( (i*j) > 1000 ){
A[i,j] <- i/j } else A[i,j] <- i*j}}

# slightly better style
for(i in 1:50){
   for(j in 1:50){
      if( (i*j) > 1000 ){
         A[i,j] <- i/j
      } else{
         A[i,j] <- i*j}}}

# good style
for(i in 1:50)
{
   for(j in 1:50)
   {
      if( (i*j) > 1000 )
      {
         A[i,j] <- i/j
      }  else
      {
         A[i,j] <- i*j
      }
   }
}
```

Some basic rules of thumb are

- when you begin a new control structure (`for`, `while if`, etc...) place the brace on the next line indented the same amount as the beginning of the control statement
- everything inside the control structure should be indented so it is clear how far in the code is nested (3 spaces is a good amount to indent)
- indent the same amount for every control structure within one program

1 Downloading and using non-default packages in R

Sometimes very useful R functionality is not included in the default packages and you need to download one of the hundreds of non-default packages submitted by various authors to R. We will demonstrate by installing the `RSQLite` package which the database handling functionality you need for the homework this week.

Packages can be installed by use of the `install.packages` command, which takes an argument that is the name of the package to be installed. In this example you would type

```
install.packages("RSQLite", dependencies=T)
```

The `dependencies=T` tag here tells R that if there are other packages that `RSQLite` needs in order to run, install those as well. Once you type this command, R will ask you which *mirrors* you would like to use.Mirrors are basically libraries that contain all publicly available R packages and allow you to download from them. When prompted to choose a mirror you should choose one close by– the closest one is located in Houghton, MI at Michigan Tech.

To use a non-default package that you have installed you must type `library(packagename)` at the beginning of each session you are using the package. To use the `RSQLite` package we just installed you would type

```
library(RSQLite)
```

and now all functions included in `RSQLite` are available to us

2 Accessing and using databases in R

Databases are pieces of software "wrapped around" a number of datasets. Essentially, you can think of a database as a list of datasets that allow you to dynamically access different datasets or parts thereof without having the store all of them in R's memory at one time. If you ever try to read in a massive dataset (say, 10^6 rows and 100 columns) into R you will see why databases are necessary.

2.1 Reading in and initializing databases in R

R has the capability to connect with a number of different types of databases but we will only be talking about how to connect with SQLite databases. This will require a non-default package in R called `RSQLite`, which we just downloaded. Load the library if you have not already done so above.

We will be using the baseball database available on ctools, which contains various pieces of information on all professional baseball players from 1884 to 2003 (a **massive** data set)– download the database and place it in your current directory. First we specify that we are using the SQLite database handler to connect to the database called `baseball.db`:

```
# the database driver is called SQLite
drive <- dbDriver("SQLite")

# establish a connection with the baseball database
connect <- dbConnect(drive, "baseball.db")
```

To see the various tables contained in the database we can use the `dbListTables` command:

```
dbListTables(connect)
```

Each of the names listed after calling the function above are the nams of the distinct datasets contained in the database. To see the variables contained in a particular datasets you can use the `dbListFields` command. For example, to see what variables are stored in the `"Salaries"` table you would type

```
dbListFields(connect, "Salaries")
[1] "yearID"   "teamID"   "lgID"     "playerID" "salary"
```

To obtain one entire table you use the `dbReadTable` command. To obtain the entire `"Salaries"` and assign it to the variable `Salaries` you type

```
Salaries = dbReadTable(connect, "Salaries")
```

Now `Salaries` is a data frame so we can carry out the usual manipulations. Suppose we were interested in the median salary over time, and want to compare salaries of AL teams vs NL teams. This can be accomplished with the following code:

```
# obtain the list of years in the dataset
Years <- unique( Salaries$yearID )

# Storage for the median salary at each year
# col 1: AL, col 2: NL
MedSal <- matrix(0, length(Years), 2)

# Loop through the years
```

```
for(year in Years)
{

   # The subset of the data corresponding to this year
   Sal.year <- Salaries[which(Salaries$yearID == year), ]

   # Not all years have data for both AL and NL. Only do the
   # median calculation for those that do
   if( length( unique(Sal.year$lgID) ) == 2 )
   {

     # which indices of this year's data correspond to AL players
     whichAL <- which(Sal.year$lgID == "AL")

     # Salaries for AL and NL players
     sal.AL <- Sal.year$salary[whichAL]
     sal.NL <- Sal.year$salary[-whichAL]

     # Calculate the median and store it in MedSal
     MedSal[which(Years==year), ] <- c( median(sal.AL), median(sal.NL) )
  }
}

# remove the rows of MedSal that don't have data
ix <- which(MedSal[,1] != 0)
Years <- Years[ix]
MedSal <- MedSal[ix,]

# plot the results
## makes the 'lines' part look better
IX <- sort( Years, index=T )
Years <- IX$x
MedSal <- MedSal[IX$ix, ]

# AL teams
plot(Years, MedSal[,1], xlab="Year", ylab="Median salary in hundred thousands",
      main="Median Salary vs. Year", axes=F, col=3, ylim=c(1e5, 1.4e6))
lines(Years, MedSal[,1], col=3)
# NL teams
points(Years, MedSal[,2], col=4)
lines(Years, MedSal[,2], col=4)

axis(1, seq(1980, 2005, by=5), seq(1980, 2005, by=5))
axis(2, seq(2e5, 1.4e6, by=2e5), seq(2, 14, by=2))
box()
```

It appears that until around 2006 the two leagues had comparable median salaries, but since then the AL has a considerably higher median.

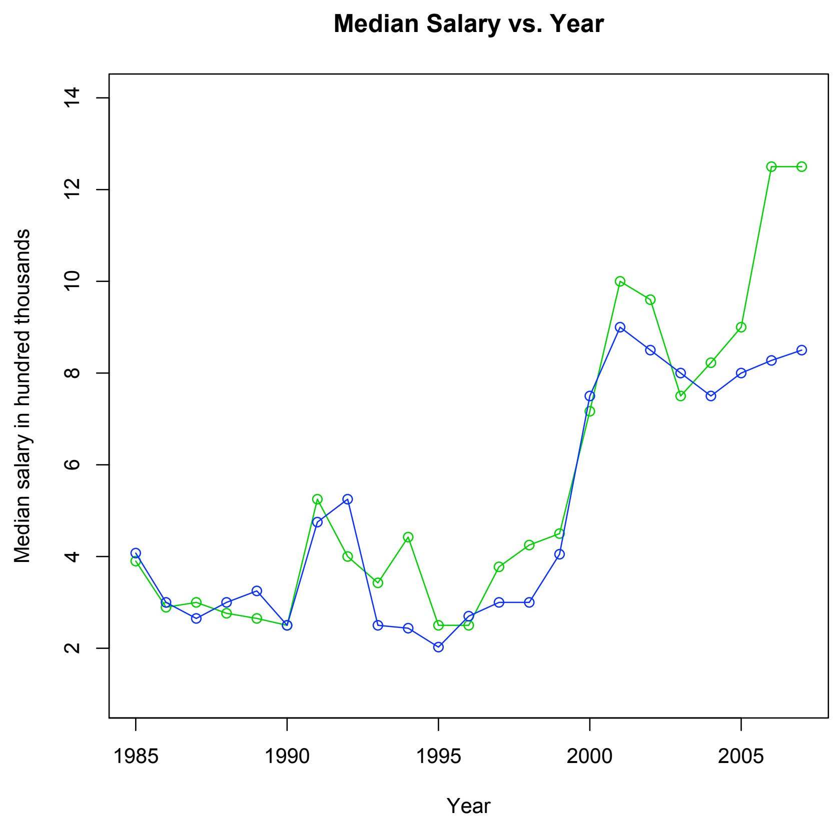

Figure 1: Median baseball salary from 1985 to 2007

2.2 Querying databases in R

Often times we do not want to full table corresponding to a particular name. Instead, we may want a dataset that is a subset of a particular table. For this we need to submit a *query*, or search, which consists of the R command `dbGetQuery(dbconnection, SQLline)`.

- The first argument is the name of your database connection, `connect` for the baseball database initialized above
- The second argument is a line of code from a database language known as SQL.

Our learning of SQL syntax is limited to combining two commands:

- `SELECT`, which indicates which tables in the database should be searched
- `WHERE`, which specifies what constraints (if any) should be obeyed when making the search

The basic psuedocode for the second argument of `dbGetQuery` is

```
"SELECT column(s) from table(s) WHERE constraint(s)"
```

We will now go through how to specify the three arguments of the line above: `column(s)`, `table(s)`, and `constraint(s)`. For the first argument, `column(s)`:

- `*`: select all columns of specified table
- otherwise, type specific column names seperated by commas.
  - If you wanted the variable names `var1`, `var2` from a specific table you would place `var1, var2` in place of `column(s)` in the pseudocode above.

For the second, argument, `table(s)`,

- You must enter a value that appears in the list `dbListTables(connect)`, without the quotes. If you want data from more than one table, the separate table names should be separated by commas.
  - Suppose one element of `dbListTables(connect)` is called `"Table1"`. Then you would type `Table1` in place of `table(s)` in the psuedocode above

For the third argument, `constraint(s)`,

- You enter logical statements involving the column names of the specified table(s). Each seperate constraint is seperated by `AND`.
- Suppose the constraint variable, `var3` is categorical. Then you specify the constraint by `var3 in ('Value1', 'Value2')`
- If the constraint variable, `var4` is numeric, then you specify the constraint with `<,>,==`. For example, `var4 > 12`.
- To combine both statements above, you would write

  ```
  var3 in ('Value1', 'Value2') AND var4>12
  ```

The combination of all the statements above would yield a `dbGetQuery` statement of the form

```
dbGetQuery(connect, "SELECT var1, var from Table1
   WHERE var3 in ('Value1', 'Value2') AND var4>12")
```

The code above does not correspond to a real database so we will do some examples with the baseball database. Suppose we wanted to extract the player ID from each AL all star in the database since 1972. What proportion of all stars in the database satisfy these conditions? What proportion of AL all stars in the database satisfy these conditions?

```
dbListTables(connect)
# Apparently all-star information is stored in the table "Allstar"

# Look at what variables are in Allstar
dbListFields(connect, "Allstar")

# Extract all player IDs from the table Allstar
D <- dbGetQuery(connect, "SELECT playerID from Allstar")
D <- unique(D)
```

```
# Extract player IDs from table Allstar for AL players
D.AL <- dbGetQuery(connect, "SELECT hofID from Allstar WHERE lgID in ('AL') ")
D.AL <- unique(D.AL)

# Extract player IDs from table Allstar for AL players after 1972
D.AL1972 <- dbGetQuery(connect,
    "SELECT playerID from Allstar WHERE lgID in ('AL') AND yearID>1972")
D.AL1972 <- unique(D.AL1972)

# Proportion of all all-stars that are AL and after 1972
nrow(D.AL1972)/nrow(D)
[1] 0.2978868

# Proportion of AL all-starts that are after 1972
nrow(D.AL1972)/nrow(D.AL)
[1] 0.5538657
```

Suppose we are interested in the proportion of players that made an all-star that eventually landed in the hall of fame. For this we can either query multiple tables simultaneously or do multiple queries of single tables. Since querying multiple tables simultaneously can take several minutes to load we will not do that here.

We will calculate this proportion by doing two separate queries:

```
# Extract IDs of each player to make an all-star game
AS <- dbGetQuery(connect, "SELECT playerID from Allstar")

# Extract IDs and whether or not they made the hall of fame for
# every player ever up for HOF nomination in the data set.
HOF <- dbGetQuery(connect, "SELECT playerID, inducted from HallOfFame")
```

Notice the hall of fame IDs are the same as player IDs, except they have the letter `h` stuck to the end, the following function will test for equivalence of a hall of fame ID and player ID (you don't need to know this)

```
is.same <- function(pID, hID) (hID == paste(pID,"h",sep=""))
```

Now we can write a program to see what proportion of all-stars eventually got accepted into the hall of fame (so far):

```
# all players who made an all-star game
AllstarIDs <- unique(AS$playerID)

# all players ever considered for HOF induction
HOFer <- unique( HOF$hofID )

# See which candidates actually got in
got.in <- rep(0, length(HOFer))
for(i in 1:length(got.in))
```

```
{

    # vector of induction statuses for each candidate
    induct <- HOF$inducted[which(HOF$hofID==HOFer[i])]

    # is one of then a Y
    got.in[i] <- ( sum(induct=="Y") > 0 )

}

# now HOFer will only contain those candidates that got in
HOFer <- HOFer[which(got.in == 1)]

MATCH <- rep(0, length(AllstarIDs))
# loop through all-stars to see if they are one of the people who got into the HOF
for(i in 1:length(AllstarIDs))
{

   # loop through the HOF IDs to see if any of them match
   for(j in 1:length(HOFer))
   {

      if( is.same( AllstarIDs[i], HOFer[j] ) )
      {

         MATCH[i] <- 1
         # once we've found a match stop looping through HOFer
         break
      }

   }

}
mean(MATCH)
[1] 0.08316292
```

The table `Master` contains various bits of information about each player in the data base.

```
dbListFields(connect, "Master")
```

Exercise: Use the previous example to extract the appropriate columns and determine

1. The proportion of players who had a nickname (stored in the variable `nameNick`)
2. The total number of players who were born in Michigan (stored in the variable `birthState`)
3. The total number of players who went to either University of Michigan or Michigan State (stored in the variable `college`)

For parts 2 and 3 you will have to first delete the NAs from the table If your table is stored in a variable `A`, then you can delete the NAs by typing:

```
A = A[-which(is.na(A)),]
```

Then `A` will be a vector with all of the non-NA values in it.