What is R?

R is a statistics package freely available at `http://www.r-project.org/`. Since **R** is a programming language, it is flexible – but that comes with the price of a somewhat steep learning curve. The best way to learn **R** is through practice.

In this introduction, we will use the command prompt > to execute the commands. The command line environment will probably not be sufficient for large-scale or even small-scale programming assignments, and so you are encouraged to compose your work using an editor. If you work on UNIX-based systems, then you might consider **Emacs**, which has an add-on called ESS (Emacs Speaks Statistics): please see `http://ess.r-project.org/` for documentation.

Getting help

Having trouble with **R**? Good news: **R** has good documentation. Suppose that that you are unsure about how to use a function called *myfun* in **R**. Try typing `help(`*myfun*`)` or `?(`*myfun*`)` at the command prompt. Of course, you can check out the web by performing a search query on Google. The internet will probably have an answer for you.

1 Getting Started: Elementary Operations, Assigning Variables, Vectors, and the Basics of Lists

In what follows, the # indicates a comment, and so everything proceeding the # key will not be compiled. You do **not** need to type in the commands after the # key, but when you turn in your programming assignments, you should nevertheless comment your work using the # symbol.

Arithmetic operations

```
> 3 + 5
[1] 8
> 8 - 6
[1] 2
> 3*5
[1] 15
> 10/2
[1] 5
> sqrt(100)
[1] 10
> exp(1)
[1] 2.718282
> log(2.718)  #natural logarithm
[1] 0.9998963
> log2(16)
[1] 4
```

Working with constants

Assigning variables lets us manage our work more easily. Notice that we can either use <- or = for assigning values.

```
> a <- 5
> b = 6
> a; b
[1] 5
[1] 6
> print(a)
[1] 5
```

† *A large portion of this lab has been adopted from Elizabeth Purdom's tutorial. This version has been updated by Rudy Angeles with input from Jacob Bien.*

Vectors and basic statistical summaries

Suppose we're interested in the vector `x`, which consists of the observations 4, 9, 12, 16, 7, 9, and 6. We might be interested in several statistical measures. The `c` in the assignment operator stands for "concatenate" or "combine."

```
> x <- c(4, 9, 12, 16, 7, 9, 6)
> length(x) #find the number of elements in x
[1] 7
> summary(x) #compute the five-number summary and the mean at the same time
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    4.0     6.5     9.0     9.0    10.5    16.0
> sum(x) #take the sum of all elements in x
[1] 63
> prod(x) #take the product of all elements in x
[1] 2612736
> cumsum(x) #compute the cumulative sums
[1]  4 13 25 41 48 57 63
> mean(x)
[1] 9
> median(x)
[1] 9
> sd(x)
[1] 4
> quantile(x, c(.25, .40, .75))  #empirical quantiles:  note that we can specify more than once at a time
 25%  40%  75%
 6.5  7.8 10.5
> which.min(x) #element with smallest element
[1] 1
> which.max(x) #element with largest element
[1] 4
> sort(x) #sort from smallest to largest
[1]  4  6  7  9  9 12 16
> rank(x) #gives the ranks; ties are averaged by default; see ?rank() documentation for other options
[1] 1.0 4.5 6.0 7.0 3.0 4.5 2.0
```

We can add a constant to each element (i.e., shifting), of `x`,

```
> x + 5
[1]  9 14 17 21 12 14 11
```

or simply multiply each term by a constant (i.e., scaling):

```
> 3*x
[1] 12 27 36 48 21 27 18
```

To extract a particular element of the vector, say the 3rd element, enter `x[3]`.

```
> x[3]
[1] 12
```

Or maybe you want the 1st, 2nd, and 7th elements of `x`.

```
> x[c(1, 2, 7)]
[1] 4 9 6
```

Suppose we wanted to delete the fifth element of `x`, and then store this new vector as `y`.

```
> y <- x[-5]
> y
[1]  4  9 12 16  9  6
```

Likewise, to delete the first and seventh elements of `x`:

```
> w <- x[-c(1, 7)]
> w
[1]  9 12 16  7  9
```

Character strings

We can also define character strings by using the beginning and ending quotation marks `" "`:

```
> char <- "something"
> char
[1] "something"
```

The constant values of `a` and `b`, and the vector values of `x`, `y` and `w` that we defined above are examples of *numeric* objects. Let's see if each of the variables that we defined is of class `numeric` or `character`:

```
> is.numeric(a)
[1] TRUE
> is.numeric(x)
[1] TRUE
> is.character(b)
[1] FALSE
> is.character(char)
[1] TRUE
```

Working with sequential data: the `seq()` function

R handles sequential values quite nicely.

```
> z <- 1:10 #create a sequence of integers from 1 to 10
> z
 [1]  1  2  3  4  5  6  7  8  9 10
> is.vector(z) #check to see if z is of class vector
[1] TRUE
> is.numeric(z) #of course, z is numeric data
[1] TRUE

> seq(from=10, to=20, by=2) #create a sequence from 10 to 20 in increments of 2
 [1] 10 12 14 16 18 20

> seq(0, 1, .1) #create a sequence from 0 to 1.0 in increments of .1
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

> seq(0, 1, .05)
 [1] 0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55 0.60 0.65 0.70 0.75
[17] 0.80 0.85 0.90 0.95 1.00
> seq(0, 1, length=7)
[1] 0.0000000 0.1666667 0.3333333 0.5000000 0.6666667 0.8333333 1.0000000
> ?seq() #get help on the seq() function
```

Listing already defined variables via the `ls()` function

To see all of the variables that you have defined, use the `ls()` function, which lists, in alphabetical order, all the variables you've created:

```
> ls()
[1] "a"    "b"    "char" "w"    "x"    "y"    "z"
```

Using the `rep()` function

The `rep()` function is also handy for creating patterned data:

```
> rep(1, 10) #create a vector of ten 1's
 [1] 1 1 1 1 1 1 1 1 1 1

> rep(c(1,2,3), 5) #repeat the sequence 1, 2, 3 five times
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3

> rep(c(1,2,3), each=5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3

> ?rep() #get help on the rep() function
```

Basics of lists

Oftentimes, we want a convenient and flexible way to store objects of various types and lengths, and we can do so by storing them in a *list*. To illustrate, compare:

```
> a <- c("hello", 2.3, c(1, 2)) # notice that the numbers are converted to characters and the vector is concatenated
> a
[1] "hello" "2.3"   "1"     "2"
> b <- list("hello", 2.3, c(1, 2)) # all is preserved
> b
[[1]]
[1] "hello"

[[2]]
[1] 2.3

[[3]]
[1] 1 2
```

Use double brackets to extract individual elements:

```
> b[[3]]
```

You can name elements and use the dollar sign (`$`) to retrieve by name:

```
> b <- list(greeting="hello", cost=2.3, numbers=c(1, 2))
> names(b)
[1] "greeting" "cost"     "numbers"
> b$cost
[1] 2.3
```

2 More on Vectors and Matrices

Section 1 was a quick, but easy, introduction to **R**. Like MATLAB, **R** is a vector and matrix based language, so when dealing with computations, avoid using `for()` loops as much as possible!

More on vector manipulations

Now define the new vectors `aa` and `bb` as follows:

```
> aa <- c(-3, 4, 7, 0, 6)
> bb <- c(5, 8, 20, -30, -7)
> aa+bb
[1]   2  12  27 -30  -1
> aa*bb #notice element-wise multiplication
[1] -15  32 140   0 -42
```

Now suppose that we want the dot product between `aa` and `bb`; that is, we want to compute $\sum_{i=1}^{5}(aa)_i(bb)_i$:

```
> sum(aa*bb)
[1] 115
```

Suppose we wanted 2×5 matrix `A`, whose rows are the vectors `aa` and `bb`:

```
> A <- rbind(aa, bb) #row bind
> print(A)
   [,1] [,2] [,3] [,4] [,5]
aa   -3    4    7    0    6
bb    5    8   20  -30   -7
```

Let's give the columns of `A` the names `apple, banana, mango, peach,` and `pear`.

```
> colnames(A) <- c("apple", "banana", "mango", "peach", "pear")
> print(A)
   apple banana mango peach pear
aa    -3      4     7     0    6
bb     5      8    20   -30   -7
```

Or what if we wanted a 5×2 matrix `B`, whose columns are the vectors `aa` and `bb`?

```
> B <- cbind(aa, bb) #column bind
> print(B)
     aa  bb
[1,] -3   5
[2,]  4   8
[3,]  7  20
[4,]  0 -30
[5,]  6  -7
```

Likewise, we can give names to the rows of `B`:

```
> rownames(B) <- c("Art", "Susan", "Trevor", "Rob", "Jerry")
> print(B)
       aa  bb
Art    -3   5
Susan   4   8
Trevor  7  20
Rob     0 -30
Jerry   6  -7
```

Now suppose that `Art`, `Susan`, `Trevor`, `Rob`, and `Jerry` are assigned alpha-numeric IDs:

```
> IDs <- c("A1", "B2", "C3", "D4", "E5")
> is.character(IDs) #remember that IDs of class character
[1] TRUE
> is.vector(IDs) #it's also a vector
[1] TRUE
```

Removing objects via the `rm()` function

Let's say that we're done using the variables `A` and `B`, and we want to remove them from **R**'s memory:

```
> rm(list=c("A", "B"))
> A
Error: object "A" not found
```

Factors

This type of object allows **R** to deal with categorical variables. The different possible categories are called *levels*.

```
> fac <- factor(c(2,1,2,2,4,-1), labels=c("A","B","C","D"))
> fac
[1]C B C C D A Levels: A B C D
> levels(fac)
[1]"A" "B" "C" "D"
```

Data frames

Now let's say we wanted to create a *data frame*, which is an array that allows us to mix both numeric and character data.

```
> myframe <- data.frame(aa, bb, IDs)
> myframe
  aa  bb IDs
1 -3   5  A1
2  4   8  B2
3  7  20  C3
4  0 -30  D4
5  6  -7  E5
```

Matrices

From here on out, you're encouraged to use an editor.

Consider the following two matrices:

$$A = \begin{bmatrix} 3 & -5 & 1 \\ 1 & 1 & 1 \\ -1 & 5 & -2 \\ 3 & -7 & 8 \end{bmatrix}; \quad B = \begin{bmatrix} -1 & 6 & 6 \\ 3 & -8 & 3 \\ 1 & -2 & 6 \\ 1 & -4 & -3 \end{bmatrix}$$

Here are a couple of ways to define `A` and `B` into **R**:

```
> A <- matrix(c(3, 1, -1, 3, -5, 1, 5, -7, 1, 1, -2, 8), nrow=4, ncol=3)
> A
     [,1] [,2] [,3]
[1,]    3   -5    1
[2,]    1    1    1
[3,]   -1    5   -2
[4,]    3   -7    8
```

By default, **R** reads in the data by columns, such as when we entered the matrix `A` as above. Alternatively, we could specify the option `byrow=TRUE` and enter the rows of the matrix first, as in the following example for the matrix `B`:

```
> B <- matrix(c(-1, 6, 6, 3, -8, 3, 1, -2, 6, 1, -4, -3), nrow=4, ncol=3, byrow=TRUE)
> B
     [,1] [,2] [,3]
[1,]   -1    6    6
[2,]    3   -8    3
[3,]    1   -2    6
[4,]    1   -4   -3
```

Of course, if you need any help with the `matrix()` function, invoke the help menu:

```
> ?matrix()
```

Extract the first row of `A`:

```
> A[1,]
[1]  3 -5  1
```

Now extract the second and third columns of `B`:

```
> B[,c(2,3)]
     [,1] [,2]
[1,]    6    6
[2,]   -8    3
[3,]   -2    6
[4,]   -4   -3
```

Since `A` and `B` are of the same dimension, we can add them:

```
> A + B
     [,1] [,2] [,3]
[1,]    2    1    7
[2,]    4   -7    4
[3,]    0    3    4
[4,]    4  -11    5
```

Elementwise arithmetic on matrices

Basic arithmetic calculations act on each element of the matrix. For example:

```
> A^2
     [,1] [,2] [,3]
[1,]    9   25    1
[2,]    1    1    1
[3,]    1   25    4
[4,]    9   49   64
> abs(A) #absolute value
     [,1] [,2] [,3]
[1,]    3    5    1
[2,]    1    1    1
[3,]    1    5    2
[4,]    3    7    8
```

As we did in the vector case, we can multiply `A` and `B` **elementwise**:

```
> A*B
     [,1] [,2] [,3]
[1,]   -3  -30    6
[2,]    3   -8    3
[3,]   -1  -10  -12
[4,]    3   28  -24
```

Be careful! The asterisk `*` is used for **componentwise multiplication**. In this example, we know that we cannot apply matrix multiplication to `A` and `B`.

Matrix multiplication

But we can multiply A and B^T. We carry out matrix multiplication by using the operator `%*%`:

```
> B.trans <- t(B)   #take the transpose of B
> A %*% B.trans   #multiply A by B-transpose
     [,1] [,2] [,3] [,4]
[1,]  -27   52   19   20
[2,]   11   -2    5   -6
[3,]   19  -49  -23  -15
[4,]    3   89   65    7
> A %*% B
Error in A %*% B : non-conformable arguments
```

See what happens when we try to apply matrix multiplication between A and B! **R** warns us!

Efficient statistical calculations on matrices: using the `apply()` and `scale()` functions

Suppose that A was a hypothetical data set consisting of 4 observations (rows) and 3 variables (columns). Let's compute the mean for each of the three variables of A, and the sample variances for each of the four rows of B:

```
> apply(A, 2, mean) #the 2 corresponds to the columns
[1]  1.5 -1.5  2.0
> apply(B, 1, var) #1 corresponds to the rows
[1] 16.33333 40.33333 16.33333  7.00000
```

We can also subtract the column means from each element of `A` so that each column has mean zero:

```
> scale(A, center=TRUE, scale=FALSE)
     [,1] [,2] [,3]
[1,]  1.5 -3.5   -1
[2,] -0.5  2.5   -1
[3,] -2.5  6.5   -4
[4,]  1.5 -5.5    6
attr(,"scaled:center")
[1]  1.5 -1.5  2.0
```

Let's center and scale `B` so that its columns have zero mean and unit variance:

```
> scale(B)
          [,1]       [,2]       [,3]
[1,] -1.224745  1.3587324  0.7071068
[2,]  1.224745 -1.0190493  0.0000000
[3,]  0.000000  0.0000000  0.7071068
[4,]  0.000000 -0.3396831 -1.4142136
attr(,"scaled:center")
[1]  1 -2  3
attr(,"scaled:scale")
[1] 1.632993 5.887841 4.242641
> apply(scale(B), 2, mean)
[1]  0.000000e+00 -2.775558e-17  0.000000e+00
> apply(scale(B), 2, var)
[1] 1 1 1
```

QR decomposition

Recall from numerical linear algebra that if A is an $m \times n$ matrix with linearly independent columns, then A can be factored as $A = QR$, where:

- Q is an $m \times n$ matrix whose columns form an orthonormal basis for the column space of A;
- R is an $n \times n$ upper triangular invertible matrix with positive entries on its diagonal.

For the matrix `A` that we defined, let's compute its QR-factorization and store it as the object `qr.of.A`.

```
> qr.of.A <- qr(A)
> qr.of.A
$qr
           [,1]       [,2]      [,3]
[1,] -4.4721360  8.9442719 -6.708204
[2,]  0.2236068 -4.4721360  2.236068
[3,] -0.2236068  0.7007458 -4.472136
[4,]  0.6708204 -0.3133831  1.000000

$rank
[1] 3

$qraux
[1] 1.670820 1.640895 1.000000
```

```
$pivot
[1] 1 2 3

attr(,"class")
[1] "qr"
```

Notice that `qr.of.A` contains various elements. In fact, the output of `qr.of.A` indicates that it is an object of the *class* called `qr`. That is,

```
> is.qr(qr.of.A)
[1] TRUE
```

Obtaining object information using `str()`, `names()`, `class()`, `attributes()`, and `summary()`

The class `qr` has a lot of "structure," and we can obtain information about the object `qr.of.A` using the `str()`, or *structure*, function:

```
> str(qr.of.A)
List of 4
 $ qr   : num [1:4, 1:3] -4.472 0.224 -0.224 0.671 8.944 ...
 $ rank : int 3
 $ qraux: num [1:3] 1.67 1.64 1
 $ pivot: int [1:3] 1 2 3
 - attr(*, "class")= chr "qr"
```

What does all of this output mean? Note the following:

- `qr.of.A` is a list comprising of the four elements named `qr`, `rank`, `qraux`, and `pivot`. The `$` sign prior to each element's name indicates that each element can be extracted by the `$` sign. **Important Remark! Not all objects have elements that can be extracted by the `$` sign. The elements of some S4 objects are actually extracted by the `@` sign.**

  For instance, the rank of A can be extracted by typing:

  ```
  > qr.of.A$rank
  [1] 3
  ```

- Its *names* can be easily obtained by:

  ```
  > names(qr.of.A)
  [1] "qr"    "rank"  "qraux" "pivot"
  ```

- Even if you didn't know `qr.of.A`'s *class*, simply type:

  ```
  > class(qr.of.A)
  [1] "qr"
  ```

- The `names` and `class` comprise `qr.of.A`'s so-called *attributes*:

  ```
  > attributes(qr.of.A)
  $names
  [1] "qr"    "rank"  "qraux" "pivot"

  $class
  [1] "qr"
  ```

- A good idea is also to obtain the `summary` of your new object:

  ```
  > summary(qr.of.A)
        Length Class  Mode
  qr    12     -none- numeric
  rank   1     -none- numeric
  qraux  3     -none- numeric
  pivot  3     -none- numeric
  ```

Now let's extract the `qr` element of `qr.of.A`:

```
> qr.of.A$qr
           [,1]       [,2]      [,3]
[1,] -4.4721360  8.9442719 -6.708204
[2,]  0.2236068 -4.4721360  2.236068
[3,] -0.2236068  0.7007458 -4.472136
[4,]  0.6708204 -0.3133831  1.000000
```

The `qr` "factorization" looks a little funny. In fact, the upper right triangle of `qr` is exactly the upper triangular matrix R in the QR-decomposition, whereas the remaining elements are not as clear. To actually see what the Q and R parts of our original matrix A is, type:

```
> qr.Q(qr.of.A)
           [,1]       [,2]       [,3]
[1,] -0.6708204 -0.2236068  0.6708204
[2,] -0.2236068 -0.6708204 -0.2236068
[3,]  0.2236068 -0.6708204 -0.2236068
[4,] -0.6708204  0.2236068 -0.6708204
> qr.R(qr.of.A)
          [,1]      [,2]      [,3]
[1,] -4.472136  8.944272 -6.708204
[2,]  0.000000 -4.472136  2.236068
[3,]  0.000000  0.000000 -4.472136
```

Now verify that $A = QR$ (although in general, there may be some numerical roundoff error):

```
> qr.Q(qr.of.A) %*% qr.R(qr.of.A)
     [,1] [,2] [,3]
[1,]    3   -5    1
[2,]    1    1    1
[3,]   -1    5   -2
[4,]    3   -7    8
```

Symmetric matrices

Consider the following symmetric matrix S:

$$S = \begin{bmatrix} 3 & 0.2 & 1 \\ 0.2 & 4 & 3 \\ 1 & 3 & 9 \end{bmatrix}$$

Define the matrix S in **R**:

```
> S <- matrix(c(3, .2, 1, .2, 4, 3, 1, 3, 9), nrow=3, ncol=3)
> diag(S)
[1] 3 4 9
> sum(diag(S))
[1] 16
```

We can compute its eigenvalues and eigenvectors. The eigenvectors are sorted from the largest eigenvalue to the smallest eigenvalue.

```
> eigen.of.S <- eigen(S)
> eigen.of.S
$values
[1] 10.535430  3.009737  2.454833

$vectors
           [,1]       [,2]       [,3]
[1,] -0.1304399  0.8571370 -0.4982987
[2,] -0.4169126 -0.5034171 -0.7568058
[3,] -0.8995384  0.1090293  0.4230169

> sum(eigen.of.S$values) #this should be the trace!
[1] 16
```

3 Reading in Data

Of course, in most real cases we're interested in a larger data set.

- For comma separated value (`.csv`) files, use the command `read.csv` to load the dataset into **R**.
- We can load other text files using the command `read.table`.

Monarch butterfly data set

The data set `monarch.txt` contains information from migratory Monarch butterflies which were spotted in Pismo Beach on the Central Coast of California. Let's read in that data set and call it `butterflies`. Note that we specify the option of `header=TRUE` since the raw data set already has variable names in the leading row.

```
> butterflies <- read.table("http://www-stat.stanford.edu/~rangeles/305/monarch.txt", header=TRUE)
> butterflies
   sex wetwt drywt drythx dryabd wingarea wingspan
1 Male   643   302     85    137     37.2     10.9
2 Male   590   275     67    146     33.5     10.3
3 Male   465   219     62    100     31.2     10.0
4 Male   613   321     77    131     32.3     10.0
...
> is.data.frame(butterflies)
[1] TRUE
> names(butterflies)
[1] "sex"      "wetwt"    "drywt"    "drythx"   "dryabd"   "wingarea" "wingspan"
> mean(butterflies$wetwt)
[1] 519.1864
> sd(butterflies$dryab)
[1] 35.09023
> quantile(butterflies$drywt, probs=c(.10, .90)) #compute 10th & 90th percentiles of the sample data
  10%   90%
186.0 314.2
```

It's cumbersome to always invoke the `$` sign to extract a particular variable name from the data frame `butterflies`. To get around this, `attach` this data frame into R:

```
> attach(butterflies)
> min(wingarea) #no need for the $ sign anymore, because we attached the data set
[1] 20.3
```

Now let's create two subsets of data, one for the males and one for the females.

```
> males <- butterflies[sex == "Male",]
> females <- butterflies[sex == "Female",]
> mean(males[,-1]) #can't obtain a mean for the sex variable
    wetwt     drywt    drythx    dryabd  wingarea  wingspan
541.17431 253.16514  73.56881 111.19266  33.37982  10.19633
> mean(females[,-1]) #can't obtain a mean for the sex variable
     wetwt      drywt     drythx     dryabd   wingarea   wingspan
497.594595 239.684685  65.090090 111.936937  30.804505   9.918018
> tapply(wetwt, sex, mean)
  Female     Male
497.5946 541.1743
```

Basic hypothesis testing: the t-test

Now conduct a two-sample t-test at the $\alpha = .01$ significance level to see if the overall mean wet weights between males and females are different. That is, we want to conduct the following hypothesis test:
$H_0 : \mu_{\text{male}} = \mu_{\text{female}}$
$H_A : \mu_{\text{male}} \neq \mu_{\text{female}}$

```
> t.test(wetwt ~ sex, conf.level=.99)

        Welch Two Sample t-test

data:  wetwt by sex t = -3.8558, df = 208.759, p-value = 0.0001536
alternative hypothesis: true difference in means is not equal to 0
99 percent confidence interval:
 -72.96109 -14.19834
sample estimates: mean in group Female   mean in group Male
                              497.5946             541.1743
> ?t.test #if you want to look at the documentation for t.test
```

So as seen from above, males and females have significantly different mean wet weights. Notice that the 99% confidence interval (C.I.) for the difference is $(-72.96, -14.20)$: this is the C.I. for the difference in the means between females and males. **Remark:** **R** first computes the means in alphabetical order, and hence we have females before males.

Obstacle data set

Download the file `obstacle.txt` from `http://www-stat.stanford.edu/~rangeles/305` onto to your computer. Notice this data does not have names, so when you read in the data set, you should specify `header = FALSE`. Identify the path name/directory where you downloaded this file, and read in the data set into **R**. Here's an example (yours will be different):

```
> obstacle <- read.table("C:\\Documents and Settings\\Statistician\\Desktop\\305\\obstacle.txt", header=FALSE)
```

Warning! You must specify two backward slashes \\, as opposed to one backward slash \ between levels of the path name. Alternatively, you could have used one forward slash /.

Exercise. Conduct a two-sample t-test at the $\alpha = .05$ significance level to see if the average obstacle course running times between freshmen (first column) is different from sophomores (second column). [**Hint:** Look at help menu for `t.test`. The set-up of this data set is a little different from when we did the previous t-test.]

Current working directory: `getwd()` and `setwd()`

If you need to know where your current work is being saved, type in `getwd()` in order to "get" your working directory. Moreover, if you need to change where you want **R** to save your work, specify your appropriate folder in `setwd()`.

Attaching data sets from libraries

Data sets are sometimes provided in **R** *libraries*. We need to call each library that is used for our analysis. Libraries don't only come with data sets – they come with nice (free) functions that people have written for us!

```
> library(MASS)
> data(UScereal)
> names(UScereal)
 [1] "mfr"       "calories"  "protein"   "fat"       "sodium"    "fibre"     "carbo"     "sugars"
 [9] "shelf"     "potassium" "vitamins"
> attach(UScereal)
> UScereal[mfr=="G" | mfr=="K", 1:3] #extract first three columns of UScereal where mfr=="G" or mfr=="K"
                          mfr  calories   protein
All-Bran                    K 212.12121 12.121212
All-Bran with Extra Fiber   K 100.00000  8.000000
Apple Cinnamon Cheerios     G 146.66667  2.666667
Apple Jacks                 K 110.00000  2.000000
...
```

You can also find the indices directly using `which`:

```
> which(shelf==2 & sugars>5)
 [1]5  9 11 13 16 17 22 24 27 29 33 38 39 42 49 56 62
```

Exercise. Try to answer the following questions:

- How many cereals with manufacturer `G` have more than 50 calories?
- What does the following command `%in%` do?

```
> c("G","F") %in% mfr
> mfr %in% c("N","K")
```

4 Plots

Basic plots

Try each of the following elementary plots with the `UScereal` data.

```
> hist(calories) #histogram
> plot(calories, fat) #form a x-y scatterplot
> boxplot(potassium ~ mfr) #boxplot
> plot(mfr, potassium) #form a boxplot by mfr, which is of class factor
> plot(as.numeric(mfr), potassium) #no longer a boxplot, since coerced mfr to be numeric
```

Adding to plots

The following functions add to an existing plot:

- `lines`
- `points`
- `curve`
- `rect`
- `segments`
- `abline`

You must have already set up a plotting command with a function such as `plot` or `hist` to use these commands. You can set up the coordinates/axes/range without actually plotting anything using the option `type="n"`.

```
> plot(sodium, potassium, type="n")  #set-up an empty plotting screen
> points(sodium[mfr=="G"], potassium[mfr=="G"],col="red") #add G points in red
> points(sodium[mfr=="K"], potassium[mfr=="K"],col="blue") #add K points in blue
> abline(h=c(max(potassium)-1, min(potassium)+1), lty=c(2,3))
```

Exercise. Use the `butterflies` data set to plot `wetwt` on the x-axis and `drywt` on the y-axis. Label the females by the color `red`, and label the males with the color `blue`.

Prettying your graph *(see* `help(par)` *for more details)*

You will sometimes find these are quite easy to implement, but other times some of the settings don't want to work with the plotting function you are using. It takes a good bit of experimenting.

- `par(mfrow=c(x,y))`: creates a grid of plots with `x` rows and `y` columns.

Most are options that you put in the plot command just for a particular plot.

- `xlab, ylab`: x-axis or y-axis labels
- `lty`: Line type (0=blank, 1=solid, 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings 'blank", "solid", "dashed", "dotted", "dotdash", '"longdash"', or "twodash", where "blank"' uses "invisible lines" (i.e., doesn't draw them).
- `pch`: Style of points in graph
- `bg, col.lab, col.main, col.sub`: The color for the background, labels, main title, and subtitle respectively. Usually use values like "red" or "tan" to pick color. Type `colors()` to see all options.

- `las`: Style of axis labels. (0=parallel, 1=all horizontal, 2=all perpendicular to axis, 3=all vertical)
- `font`: 1=plain, 2=bold, 3=italic, 4=bold italic

You can also set some of these things after you have already made your main plot.

- `title()`
- `axis()`

```
> par(mfrow=c(1,2))
> plot(carbo,fibre,pch=3, las=1,main="Fiber versus Carbohydrates", sub="A cool subtitle is useful")
> hist(calories, xlab="Calories", main="",sub="",col.lab="blue")
> title(main="A Histogram of my Making", sub="A Different Histogram from the Default")
```

5 Simple Linear Regression

Stat 305 is mainly concerned with regression. Before fitting a linear model, it is always good practice to plot the data first. **Do not just blindly fit a model!!!**

More on butterflies

The ecologists from the butterfly study were interested in seeing how well `drywt` could be predicted from the `wetwt`. Of course, we should plot the data first (left hand panel of Figure 1). Indeed, the scatterplot suggests the strong presence of a linear relationship.

```
> plot(wetwt, drywt, main="Plot of dry-weight vs. wet-weight")
```

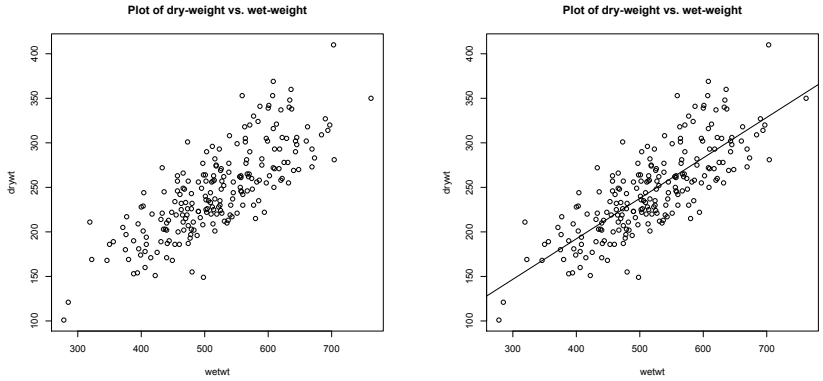


FIGURE 1: *Left panel:* Scatterplot of `drywt` as a function of `wetwt`. *Right panel:* The fitted simple linear regression line is superimposed on the scatterplot by using the `abline()` function.

Covariance and correlation: `cov()` and `cor()`

The butterfly data set consists of variables that are almost entirely numeric: the `sex` found in the first column is, of course, categorical. We can transform this data frame into a numeric matrix by removing the first column. We can then calculate the pairwise covariances and correlations:

```
> X.butterflies <- as.matrix(butterflies[,-1])  #create numeric matrix by removing sex in the first column
> cov(X.butterflies)
              wetwt      drywt     drythx      dryabd   wingarea   wingspan
wetwt    7445.57698 3383.25612 696.087090 1885.911893 208.758454 31.8886198
drywt    3383.25612 2497.42881 386.058115 1594.066418 113.115816 15.9316314
drythx    696.08709  386.05812 115.339643  187.710668  32.060050  4.6055791
```

```
dryabd   1885.91189 1594.06642 187.710668 1231.324097  54.206133  7.4553030
wingarea  208.75845  113.11582  32.060050   54.206133  10.942858  1.5082667
wingspan   31.88862   15.93163   4.605579    7.455303   1.508267  0.2520654
> cor(X.butterflies)  #Pearson correlation by default
             wetwt     drywt    drythx    dryabd  wingarea  wingspan
wetwt    1.0000000 0.7845834 0.7511473 0.6228537 0.7313571 0.7360883
drywt    0.7845834 1.0000000 0.7193112 0.9090205 0.6842446 0.6349757
drythx   0.7511473 0.7193112 1.0000000 0.4980965 0.9024216 0.8541583
dryabd   0.6228537 0.9090205 0.4980965 1.0000000 0.4669785 0.4231774
wingarea 0.7313571 0.6842446 0.9024216 0.4669785 1.0000000 0.9081470
wingspan 0.7360883 0.6349757 0.8541583 0.4231774 0.9081470 1.0000000
```

The pairwise correlation of $r = .785$ between `wetwt` and `drywt` seems moderately strong. A test of significance can obtained using the `cor.test()` function:

```
> cor.test(wetwt, drywt)

Pearson's product-moment correlation

data:  wetwt and drywt
t = 18.6836, df = 218, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.7278464 0.8306515
sample estimates:
      cor
0.7845834

> .7845834^2  #should be R^2 from simple linear regression below
[1] 0.6155711
```

Fitting linear models: the `lm()` object

Now let's fit this simple linear regression model. The `lm()` function is the workhorse of linear model fitting. Note that it creates an object of class `lm()`.

```
> butterfly.lm <- lm(drywt ~ wetwt)  #IMPORTANT:  lm(y ~ x); i.e., y as a function of x
```

To avoid confusion in a later example, we'll call our linear model `butterfly.lm` the *fitted object*, or simply "fit." Simply invoking the command `summary()` will print a convenient summary of this "fit":

```
> summary(butterfly.lm)

Call:
lm(formula = drywt ~ wetwt)

Residuals:
    Min      1Q  Median      3Q     Max
-87.737 -20.736  -2.916  19.526  88.545

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 10.44633   12.79943   0.816    0.415
wetwt        0.45440    0.02432  18.684   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 31.06 on 218 degrees of freedom
Multiple R-Squared: 0.6156,     Adjusted R-squared: 0.6138
F-statistic: 349.1 on 1 and 218 DF,  p-value: < 2.2e-16
```

Unsurprisingly, the coefficient for `wetwt` is highly significant (p-value $< 2 \times 10^{-16}$). Moreover, we can add the fitted regression line onto our scatterplot in the left hand panel of Figure 1. The superimposed simple linear regression line is displayed on the right panel Figure 1 and is generated by the following **R** code:

```
> abline(butterfly.lm)  #make sure scatterplot window is still on
```

More on `lm()` object information

There is so much information that is both calculated and stored by calling `lm()`:

- The *fitted object*, or *fit*, is stored by invoking the `lm()` command. In our example, `butterfly.lm` is our fit. This fit is itself a list, whose names are:

  ```
  > names(butterfly.lm)
   [1] "coefficients"  "residuals"     "effects"       "rank"          "fitted.values" "assign"
   [7] "qr"            "df.residual"   "xlevels"       "call"          "terms"         "model"
  ```

  and whose elements can be obtained via the `$` sign:

  ```
  > butterfly.lm$coefficients
  (Intercept)       wetwt
   10.4463330   0.4543981
  ```

- **Now this is the confusing part!** As seen above, invoking `summary()` will print a report of our fit. But you can also create a *summary object*, which is itself a list! For instance:

  ```
  > butterfly.summary <- summary(butterfly.lm)
  > names(butterfly.summary)
   [1] "call"          "terms"         "residuals"     "coefficients"  "aliased"       "sigma"
   [7] "df"            "r.squared"     "adj.r.squared" "fstatistic"    "cov.unscaled"
  ```

  So the "summary object" called `butterfly.summary` stores additional computations! We can therefore extract quantities from this summary object by using, of course, the `$` sign.

  ```
  > butterfly.summary$r.squared
  [1] 0.6155712
  > butterfly.summary$fstatistic
     value     numdf     dendf
  349.0751    1.0000  218.0000
  ```

6 Singular Value Decomposition

Let X be an $n \times p$ matrix, and assume that X has full column rank so that $\text{rank}(X) = p$. The singular value decomposition (SVD) is the most widely used matrix factorization tool in applied statistics. A key fact from numerical linear algebra states that we can factor $X = UDV^T$, where:

- U is an $n \times p$ orthogonal matrix satisfying $U^TU = I_p$;
- D is a $p \times p$ diagonal matrix consisting of the singular values of X;
- V is a $p \times p$ orthogonal matrix satisfying $V^TV = I_p$.

`scordata` from Mardia, Kent, and Bibby

The following data is an 88×5 matrix. There are 88 students, each of whom took exams in the five areas of mechanics, vector calculus, algebra, analysis, and statistics.

```
> scordata <- read.table("http://www-stat.stanford.edu/~rangeles/305/scordata.txt")
> attach(scordata)
> print(scordata[1:5,])
     X..1. X..2. X..3. X..4. X..5.
[1,]    77    82    67    67    81
[2,]    63    78    80    70    81
[3,]    75    73    71    66    81
[4,]    55    72    63    70    68
[5,]    63    63    65    70    63
```

A common approach in applied statistics problems, **especially when calculating the SVD,** is to standardize each variable to have zero mean and unit variance:

```
> scordata.std <- scale(scordata)
```

Now let's take the SVD of the standardized exam matrix:

```
> scordata.svd <- svd(scordata.std)
> names(scordata.svd)
[1] "d" "u" "v"
> scordata.svd$d
[1] 16.635663  8.021393  6.221894  5.809186  4.631779
> U.matrix <- scordata.svd$u
> V.matrix <- scordata.svd$v
> t(U.matrix) %*% U.matrix  #should be I_5
              [,1]          [,2]          [,3]          [,4]          [,5]
[1,]  1.000000e+00 1.301043e-16 2.775558e-17 -7.632783e-17  6.245005e-17
[2,]  1.301043e-16 1.000000e+00 5.637851e-17  1.222980e-16  1.231654e-16
[3,]  2.775558e-17 5.637851e-17 1.000000e+00  6.938894e-18  2.341877e-16
[4,] -7.632783e-17 1.222980e-16 6.938894e-18  1.000000e+00 -1.561251e-16
[5,]  6.245005e-17 1.231654e-16 2.341877e-16 -1.561251e-16  1.000000e+00
> t(V.matrix) %*% V.matrix  #should be I_5
              [,1]          [,2]          [,3]          [,4]          [,5]
[1,]  1.000000e+00  3.608225e-16 -2.775558e-16 0.000000e+00 -2.775558e-17
[2,]  3.608225e-16  1.000000e+00 -6.106227e-16 2.220446e-16 -9.714451e-17
[3,] -2.775558e-16 -6.106227e-16  1.000000e+00 2.775558e-17 -4.163336e-17
[4,]  0.000000e+00  2.220446e-16  2.775558e-17 1.000000e+00  5.551115e-17
[5,] -2.775558e-17 -9.714451e-17 -4.163336e-17 5.551115e-17  1.000000e+00
```

SVD application (i): approximating X with a rank r matrix

Although our matrix X has rank p, it may be the case that its effective rank is smaller, in the sense that a lower ranked matrix may approximate the original X entries fairly well. The full SVD of X can be written as:

$$X = UDV^T = \sum_{k=1}^{p} d_k u_k v_k^T$$

Consider the problem of minimizing the Frobenius norm:

$$\min_{\tilde{X}:\ \mathrm{rank}(\tilde{X})=r} ||X - \tilde{X}||_F^2 \quad = \quad \min_{\tilde{X}:\ \mathrm{rank}(\tilde{X})=r} \sum_{i=1}^{n}\sum_{j=1}^{p}(X_{ij} - \tilde{X}_{ij})^2$$

Linear algebra theory tells us that the global minimizer for a given rank r is

$$\underset{\tilde{X}:\ \mathrm{rank}(\tilde{X})=r}{\mathrm{argmin}} ||X - \tilde{X}||_F^2 \quad \equiv \quad \hat{X}_r = \sum_{k=1}^{r} d_k u_k v_k^T,$$

which is the rank r SVD of X. In fact, the sum of squared errors is the sum of the remaining singular values squared:

$$||X - \hat{X}_r||_F^2 = \sum_{k=r+1}^{p} d_k^2$$

We illustrate this below:

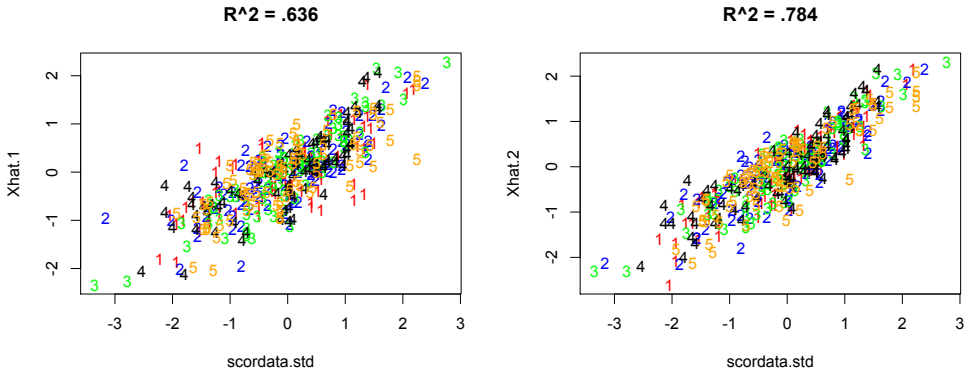


FIGURE 2: Plot of first two SVD rank approximations.

```
> Xhat.1 <- scordata.svd$d[1]*scordata.svd$u[,1]%*%t(scordata.svd$v[,1])
> Xhat.2 <- Xhat.1 + scordata.svd$d[2]*scordata.svd$u[,2]%*%t(scordata.svd$v[,2])
> sum((scordata.std - Xhat.1)^2) #rank 1 approx
[1] 158.2547
> sum(scordata.svd$d[2:5]^2)
[1] 158.2547
> sum((scordata.std - Xhat.2)^2) #rank 2 approx
[1] 93.91198
> sum(scordata.svd$d[3:5]^2)
[1] 93.91198
```

Moreover, if we were to take the sample correlation between the 88×5 entries of X_{ij} and $(\hat{X}_r)_{ij}$, we see that:

```
> # R^2
> scordata.svd$d^2/sum(scordata.svd$d^2)
[1] 0.63619603 0.14791437 0.08899303 0.07757848 0.04931810
> cor(as.vector(Xhat.1), as.vector(scordata.std))^2  #should be .636
[1] 0.636196
> cor(as.vector(Xhat.2), as.vector(scordata.std))^2  #should be .784 = .636 + .148
[1] 0.7841104
```

Let's plot the approximated $\hat{X}_1$ and $\hat{X}_2$ versus the original entries of the X matrix. Let's identify the first exam of mechanics as `1` on our plot, and plot `2, 3, 4`, and `5` for the other corresponding exams. Also, let's label each test in its own distinct color. See Figure 2.

```
> par(mfrow=c(1,2))  #1 row of graphics, 2 columns
> plot(scordata.std, Xhat.1, main="R^2 = .636", type="n") #create empty plot
> text(scordata.std, Xhat.1, labels=rep(1:5, each=88),
+      col=rep(c("red", "blue", "green", "black", "orange"), each=88))
> plot(scordata.std, Xhat.2, main="R^2 = .784", type="n") #create empty plot
> text(scordata.std, Xhat.2, labels=rep(1:5, each=88),
+      col=rep(c("red", "blue", "green", "black", "orange"), each=88))
```

SVD application (ii): principal components analysis

In some problems, it may be useful to work in a set of coordinates/variables that are *derived* from our raw data. *Principal components analysis*, or PCA, is one commonplace method. The kth principal component is defined to be Xv_k, where v_k is the kth singular vector from the SVD. Each principal component is mutually orthogonally to each other. Moreover, each component has sample variance $v_k^T \hat{\Sigma}_X v_k = d_k^2/(n-1)$, where $\hat{\Sigma}_X$ is the sample covariance matrix of X with denominator $(n-1)$, as illustrated below:

```
> pca1 <- scordata.std %*% scordata.svd$v[,1]
> pca2 <- scordata.std %*% scordata.svd$v[,2]
> cov(pca1, pca2)  #should be 0
              [,1]
[1,] -1.311087e-16
> scordata.cov.matrix <- cov(scordata.std)
> variance1 <- t(scordata.svd$v[,1]) %*% scordata.cov.matrix %*% scordata.svd$v[,1]
> variance2 <- t(scordata.svd$v[,2]) %*% scordata.cov.matrix %*% scordata.svd$v[,2]
> var(pca1)
        [,1]
[1,] 3.18098
> variance1
        [,1]
[1,] 3.18098
> scordata.svd$d[1]^2/87
[1] 3.18098
> var(pca2)
          [,1]
[1,] 0.7395718
> variance2
          [,1]
[1,] 0.7395718
> scordata.svd$d[2]^2/87
[1] 0.7395718
```

7 Writing Functions

In this section of the lab, you should open a text editor.

Basic control functions

You define a function in **R** using the command `function`. The following function returns the mean, standard deviation, and upper and lower 95% confidence interval limits in the form of a list.

```
> mysum <- function(x, conf.inv=T){
    m<-mean(x)
    if(conf.inv == T){
        n <- length(x)
        uppconf <- mean(x)+2*sd(x)/sqrt(n)
        lowconf <- mean(x)-2*sd(x)/sqrt(n)
        return(list(mean=m, sd=sd(x), uppconf=uppconf, lowconf=lowconf))
    }
    else return(list(mean=m, sd=sd(x)))
}
> mysum(potassium)
$mean [1]159.1197

$sd [1]180.2886

$uppconf [1]203.8438

$lowconf [1]114.3957
```

Basic programming functions are:

- `if`
- `else`
- `while`
- `break`
- `next`
- `for`

Note that `for()` loops are generally slow in **R**, and using `apply` or `sapply` is preferable if the function is not actually recursive. For example, the following code that finds the upper confidence interval for each

```
> my.ind <- c(2,4,8)
> x <- vector(length=length(my.ind))
> n <- nrow(UScereal)
> for(i in 1:length(my.ind)){
    x[i]<-mean(UScereal[,my.ind[i]])+2*sd(UScereal[,my.ind[i]])/sqrt(n) }
> x
[1]164.890738   1.831168  11.498387
```

could be written as:

```
> x <- apply(UScereal[,my.ind], 2, function(y){mean(y)+2*sd(y)/sqrt(length(y))})
> x
  calories        fat     sugars
164.890738   1.831168  11.498387
```

Exercise. Write a function so that for a given matrix, the function will plot the standard 95% confidence intervals for each column. Give options for line thickness and color. For example, you might name your function `myplot.CI`, and it might have the form:

```
> myplot.CI(UScereal[,c(3,4,6)], col=c("red","green","blue"), lwd=1:3)
```

8 Saving Your Workspace

Before exiting **R**, you will probably want to save your work. After all, you've created all of these variables already and probably will not want to re-compile your previous work. Depending on how much work you want to save, there are two general ways to save your workspace. The output file ends in `.RData`:

- To save your *entire* workspace (i.e., all variables that you have defined), use the command `save.image("Lab1.RData")`.
- Perhaps you want to save only a *subset* of your work, say the variables, `x, y` and `z`. Then use the command `save(x, y, z, file="Lab1Subset.RData")`

So the next time you launch **R**, all you need to do is *load* in your previous workspace. That is:

```
> load("Lab1.RData")
```

Just make sure that `"Lab1.RData"` can be found in your current working directory. Don't forget about the functions `getwd()` and `setwd()`.