Introduction to R

Matthew Thomas

Why R?

- R is free, open source, and incredibly popular
- There is a large (and welcoming) community of R programmers online who can help troubleshoot code and answer questions
- The language is incredibly well (and consistently) documented
- There are thousands of packages which implement statistical estimators and other use cases.

Defining variables/basic data types

Vectors and Assignment

The function `c()` takes vectors and creates a new longer vector. The assignment operator `<-` is a shortcut for the `assign()` function.

```
x <- c(1,2,4,6,10:13)
assign("y",c(1,2,4,6,10:13))
```

```
x
```

```
[1]  1  2  4  6 10 11 12 13
```

```
y
```

```
[1]  1  2  4  6 10 11 12 13
```

Operators

```r
x/y # Operators on vectors apply element-wise
```

```
[1] 1 1 1 1 1 1 1 1
```

```r
(1:2) * (1:8)  # Vectors will repeat if necessary
```

```
[1]  1  4  3  8  5 12  7 16
```

```r
(1:6) > (6:1)  # Logical operators: <,<=,==,>=,>,!=
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```r
!(1:6) > (6:1) # Reverse logic with !
```

```
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

Matrices and Apply

A matrix is a vector with a dimension attribute. Matrices are filled column by column unless specified.

```
(mat <- matrix(data = x, ncol = 2))
```

```
     [,1] [,2]
[1,]    1   10
[2,]    2   11
[3,]    4   12
[4,]    6   13
```

Sub-setting Matrices

You can subset a matrix using row,col indexing.

```
mat[1,]  # First row of matrix
```

```
[1]  1 10
```

```
mat[,2]  # Second column of matrix
```

```
[1] 10 11 12 13
```

```
mat[1,2] # Second element of first row
```

```
[1] 10
```

Warning about One Dimensional Matrices

An nx1 matrix and a vector are not the same thing. For example, a nx1 matrix will not replicate if necessary.

```
matrix(1:2) * matrix(1:8)
```

```
Error in matrix(1:2) * matrix(1:8): non-conformable arrays
```

Defining Functions

Functions are objects in R that can be applied to other objects. `c()`, `mean()`, and `sum()` are examples of built-in functions. You can also write your own functions.

```
sumsq <- function(var){
  return(sum(var^2))
}
```

Calling Functions

These functions can be called just as any built-in function.

```
sumsq(c(1,2))
```

```
[1] 5
```

The convenience operator %>% passes the preceding object to the first argument of any function.

```
c(1,2) %>% sumsq()
```

```
[1] 5
```

Linear Algebra and apply

The `apply()` function applies some function across rows (`MARGIN=1`) or columns (`MARGIN=2`) of a matrix.

```
apply(X=mat, MARGIN=1, FUN=sumsq)
```

```
[1] 101 125 160 205
```

The operators `%*%` and `%^%` do matrix multiplication and exponentiation. The function `t()` transposes. If you can accomplish a task with linear algebra, it is generally faster than `apply()`.

```
c(mat^2 %*% c(1,1))
```

```
[1] 101 125 160 205
```

for example is more than twice as fast for a large matrix.

Lists

Lists can contain any object types.

```
z <- list( "y" = y,
           "istwo" = y^2 == y*2,
           "p" = runif(8)*(1:4)/y^2 )
```

You can reference items from a list using brackets or dollar sign

```
z["y"]  # Returns a single element list
```

```
$y
[1]  1  2  4  6 10 11 12 13
```

```
z$istwo # Returns a vector
```

```
[1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

Dealing with data frames

Creating a data frame

You can make a data frame using vectors or a list. Data frames are special lists with elements of the same length.

```
(df1 <- data.frame(z))
```

```
   y istwo           p
1  1 FALSE 0.307766111
2  2  TRUE 0.128836251
3  4 FALSE 0.103560456
4  6 FALSE 0.006264794
5 10 FALSE 0.004685493
6 11 FALSE 0.007996210
7 12 FALSE 0.016925055
8 13 FALSE 0.008764983
```

Adding to data frames

You can reference and add to a data frame just as you can with any other list. However, data frames will repeat elements if necessary to enforce the length requirement.

```
df1$prod <- LETTERS[1:4]
head(df1)
```

```
   y istwo           p prod
1  1 FALSE 0.307766111    A
2  2  TRUE 0.128836251    B
3  4 FALSE 0.103560456    C
4  6 FALSE 0.006264794    D
5 10 FALSE 0.004685493    A
6 11 FALSE 0.007996210    B
```

Matrix-like properties of data frames

Due to the length requirement, data frames have limited matrix like properties. You can index a data frame just like a matrix.

```
df1[1,] # First row of data frame
```

```
  y istwo         p prod
1 1 FALSE 0.3077661    A
```

You can even apply most operators to **numeric** data frames. Linear algebra operators do not work on data frames.

```
df1[1,1:3]+1 # Have to exclude prod
```

```
  y istwo        p
1 2     1 1.307766
```

Manipulating data frames

You can manipulate data using the traditional list interface

```
df1$ly <- log(df1$y)
```

The `tidyverse` package has introduced another way to do this using the `mutate()` function

```
df1 <- df1 %>% mutate(ly2 = log(y))
head(df1,4)
```

```
  y istwo           p prod        ly       ly2
1 1 FALSE 0.307766111    A 0.0000000 0.0000000
2 2  TRUE 0.128836251    B 0.6931472 0.6931472
3 4 FALSE 0.103560456    C 1.3862944 1.3862944
4 6 FALSE 0.006264794    D 1.7917595 1.7917595
```

Regression

Running a regression

If you just want to run a regression in R, often do not need to manipulate data. Regressions in R allow you to adjust variables using "formulas". Suppose we want to estimate the following model:

$$\log(y) = \beta_0 + \beta_1 \log(p) + \beta_2 prodB + \beta_3 prodC + \beta_4 prodD$$

```r
lm(log(y) ~ log(p) + prod, data = df1) %>% summary()
```

```
              Estimate Std. Error    t value   Pr(>|t|)
(Intercept) -0.6884210  0.4454595 -1.5454178 0.21996050
log(p)      -0.5624574  0.1054163 -5.3355844 0.01286949
prodB        0.2996511  0.3992586  0.7505189 0.50744411
prodC        0.8391868  0.3989975  2.1032383 0.12617574
prodD        0.1079781  0.4344711  0.2485277 0.81976802
```

Interaction terms

You can add interaction terms by using a `:` between two variable names.

```
lm(log(y) ~ log(p) + log(p):prod, data = df1) %>% summary()
```

```
                Estimate Std. Error    t value  Pr(>|t|)
(Intercept)  -0.49445442 0.40942075 -1.2076926 0.3136912
log(p)       -0.51682229 0.11631625 -4.4432511 0.0212005
log(p):prodB -0.07919147 0.10955938 -0.7228178 0.5220465
log(p):prodC -0.23696729 0.11990627 -1.9762711 0.1425737
log(p):prodD -0.02477109 0.09591708 -0.2582553 0.8129138
```

Removing the constant

You can suppress the constant by adding −1 to the formula. Note that it automatically adds the dummy for product A back into the regression.

```
lm(log(y) ~ log(p) + prod - 1, data = df1) %>% summary()
```

```
         Estimate Std. Error    t value   Pr(>|t|)
log(p) -0.5624574  0.1054163 -5.3355844 0.01286949
prodA  -0.6884210  0.4454595 -1.5454178 0.21996050
prodB  -0.3887699  0.4593170 -0.8464087 0.45949760
prodC   0.1507657  0.4375467  0.3445705 0.75315800
prodD  -0.5804429  0.5889769 -0.9855105 0.39703708
```

Polynomials

R does not allow arbitrary binary operators inside of an equation.

```
lm(log(y) ~ p + p^2, data = df1) %>% summary()
```

```
             Estimate Std. Error  t value     Pr(>|t|)
(Intercept)  2.304784   0.161220 14.29590 7.328857e-06
p           -8.236559   1.302684 -6.32276 7.314177e-04
```

To run a polynomial fit, you need to use the `poly` function

```
lm(log(y) ~ poly(p,2), data = df1) %>% summary()
```

```
             Estimate Std. Error   t value     Pr(>|t|)
(Intercept)  1.702692  0.1190342 14.304225 3.009578e-05
poly(p, 2)1 -2.326491  0.3366796 -6.910105 9.728720e-04
poly(p, 2)2  0.495560  0.3366796  1.471904 2.010238e-01
```

Overriding

But what if you just want the square term? For that, you need to override using the inhibit function, `I()`.

```
lm(log(y) ~ p^2, data = df1) %>% summary()
```

```
             Estimate Std. Error  t value     Pr(>|t|)
(Intercept)  2.304784   0.161220 14.29590 7.328857e-06
p           -8.236559   1.302684 -6.32276 7.314177e-04
```

```
lm(log(y) ~ I(p^2), data = df1) %>% summary()
```

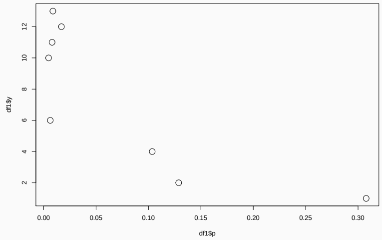
```
              Estimate Std. Error   t value     Pr(>|t|)
(Intercept)   2.073194  0.2192395  9.456295 7.959413e-05
I(p^2)      -24.189704  6.4086673 -3.774529 9.239465e-03
```

Visualization

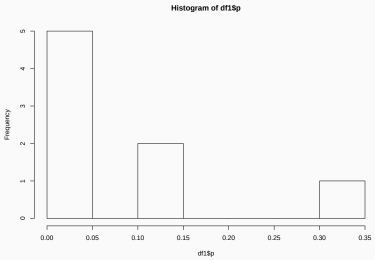
Builtin graphics

There are several basic builtin plot commands builtin to R.

```
plot(df1$y ~ df1$p, cex=2)
```



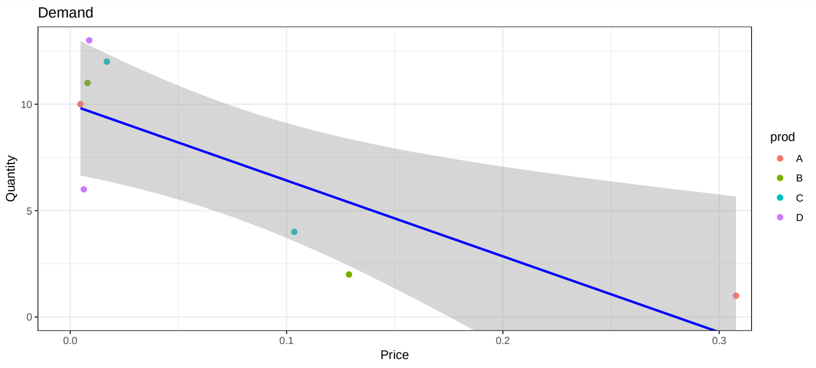

```
hist(df1$p, breaks = 8)
```



They are not very pretty, but they are very easy to use.

ggplot2 graphics

```
ggplot(data = df1, aes(x=p, y=y, col=prod)) +
  geom_point(size=2) +
  geom_smooth(method="lm", col="blue", size=1) +
  coord_cartesian(xlim=c(0,0.3), ylim=c(0,13)) +
  labs(title="Demand", y="Quantity", x="Price")
```



Appendix

Converting a data frame to a matrix

Because a matrix can contain categorical variables and strings, it is not always possible to directly convert a data frame to a matrix. An all numeric data frame can be converted by simply using `as.matrix()`

```
dfa <- data.frame(a=1:5,b=77:81,c=log(22:18))
dfb <- data.frame(a=letters[1:5],b=77:81,c=log(22:18))
```

```
as.matrix(dfa)
```

```
     a  b        c
[1,] 1 77 3.091042
[2,] 2 78 3.044522
[3,] 3 79 2.995732
[4,] 4 80 2.944439
[5,] 5 81 2.890372
```

```
as.matrix(dfb)
```

```
     a   b    c
[1,] "a" "77" "3.091042"
[2,] "b" "78" "3.044522"
[3,] "c" "79" "2.995732"
[4,] "d" "80" "2.944439"
[5,] "e" "81" "2.890372"
```

Converting a data frame to a matrix

In order to properly convert a data frame with strings or factors into a numeric matrix, we need to use `model.matrix()`. This is what R uses when it runs regressions.

```
model.matrix(~a+b+c-1,dfb)
```

```
  aa ab ac ad ae  b        c
1  1  0  0  0  0 77 3.091042
2  0  1  0  0  0 78 3.044522
3  0  0  1  0  0 79 2.995732
4  0  0  0  1  0 80 2.944439
5  0  0  0  0  1 81 2.890372
attr(,"assign")
[1] 1 1 1 1 1 2 3
attr(,"contrasts")
attr(,"contrasts")$a
```