RWorkshop

March 27, 2020

[1]:
```
#Basic data types
#Numeric
1
#Character
"string"
#Vector
c( 1, 2, 3, 4 ) #N.B. "c" is for "concatenante"


newVar = 1
newVar2 <- 10

newvar <- 2

newVar
newVar2
newvar
#Newvar
#Error: object 'Newvar' not found
#Captilization matters!

newVector <- c(6 ,2,    8 ,-9) #N.B. spacing doesn't matter, but having␣
 ↪consistent spacing looks cleaner

newVector

#Access individual elements of a vector with `[]`
newVector[1]
newVector[2]

#Access ranges using `:`

newVector[2:4]

#Accessisng elements with vector of indexes
```

```
newVector[c(1,3)]

#Negative indexes
newVector[-2]
```

```
1
'string'
1. 1 2. 2 3. 3 4. 4
1
10
2
1. 6 2. 2 3. 8 4. -9
6
2
1. 2 2. 8 3. -9
1. 6 2. 8
1. 6 2. 8 3. -9
```

[2]:
```
#"Complicated" data types:

#matrices

#Basically, a two-dimensional vector

newMatrix <- matrix(
    c(1,2,3,4,5,6,7,8),
    nrow=2
)

newMatrix

newMatrix2 <- matrix(
    c(1,2,3,4,5,6,7,8),
    nrow=4
)

newMatrix2

newMatrix3 <- matrix(
    c(1,2,3,4,5,6,7,8),
    nrow=2,
    byrow = T
)

newMatrix3
#Note the difference between newMatrix and newMatrix3

#If you have a pre-formed vector, you just pass that in to matrix()
```

```
newMatrix4 <- matrix(
    newVector,
    nrow=2
)

newMatrix4
```

A matrix: 2 Œ 4 of type dbl

1	3	5	7
2	4	6	8

A matrix: 4 Œ 2 of type dbl

1	5
2	6
3	7
4	8

A matrix: 2 Œ 4 of type dbl

1	2	3	4
5	6	7	8

A matrix: 2 Œ 2 of type dbl

6	8
2	-9

[3]:
```
#More "complicated" data types:

#data.frame

dataframe <- data.frame(
    col1 = c(1,2,3,4),
    col2 = c("a", "b", "c", "d"),
    col3 = c("Hello", "Hello", "Goodbye", "Goodbye"),
    stringsAsFactors=F #We'll get into what this means in a moment
)

dataframe

#retreive columns by `$`
dataframe$col1 #This returns a vector represented by column 1

dataframe$col2

#Dataframes are essentially matrices

#get element in row 3 column 2
dataframe[3,2]

#can use the same principles with vectors
dataframe[1:3, 2] #Rows 1 through 3 from column 2

#What if you want all rows from two columns
#We know that there are 4 rows in this data.frame
dataframe[1:4, 2]
```

```
#What if we didn't know?

dataframe[, 2] #N.B. this gives the same result as dataframe$col2

#Can do the same with columns

dataframe[1, ] #Gives row 1
```

A data.frame: 4 Œ 3

col1 <dbl>	col2 <chr>	col3 <chr>
1	a	Hello
2	b	Hello
3	c	Goodbye
4	d	Goodbye

1. 1 2. 2 3. 3 4. 4
1. ′a′ 2. ′b′ 3. ′c′ 4. ′d′
′c′
1. ′a′ 2. ′b′ 3. ′c′
1. ′a′ 2. ′b′ 3. ′c′ 4. ′d′
1. ′a′ 2. ′b′ 3. ′c′ 4. ′d′

A data.frame: 1 Œ 3

col1 <dbl>	col2 <chr>	col3 <chr>
1	a	Hello

[4]:
```
#lists

#Lists are like vectors
#Except you can hold different objects

newList <- list(
    integerType = newVar,
    vectorType = newVector,
    dataframeType = dataframe
)

#Like data.frames, you can access with indexes or `$`

newList$integerType

newList$dataframeType$col3 #access column 3 of the dataframe contained in the␣
 ↪list

#The difference of between these next two lines is somewhat semantic, but␣
 ↪important
newList[2]
newList[[2]]

typeof(newList[2])
```

```
typeof(newList[[2]]) #N.B. double is just a type of numeric, but it can be␣
 ↪signed i.e. less than zero


#Fun fact, a good majority of objects that returned from functions are, in␣
 ↪fact, lists!
#Just lists by different names
```

1
1. 'Hello' 2. 'Hello' 3. 'Goodbye' 4. 'Goodbye'
$vectorType = 1. 6 2. 2 3. 8 4. -9
1. 6 2. 2 3. 8 4. -9
'list'
'double'

[5]:
```
#Operators

#Arithmetic

print("Arithmetic operators:")
1 + 3

1 - 3

2 * 9

18 / 3

print("Power operations")
2**3
2^3

81^(1/2)
sqrt(81)

sqrt(-1) #only real numbers here -- NaN is typically a good indication that␣
 ↪something went awry
1/0 #Inf -- also a pretty good indicator of weirdness

-1/0 #Inf can be signed too

print("Modular arithmetic")
3/2
3%%2
5%/%2

#vector math
```

```
newVector * 2

vector2 <- c(1,2,4, 6)

newVector * vector2


#Strings!

#"Hello" + "World"
#Unlike in other programming langauges were the above line would automatically␣
 ↪combine the two strings, R doesn't do this
#Instead, use paste or paste0

paste("Hello", "World")
paste0("Hello", "World")
#There's a third argument for `paste` called 'sep' where you can define the␣
 ↪separating character
paste("Hello", "World", ' ')

#Vectorization
paste(as.character(newVector), "Potato")
#N.B. you can force numbers to be strings and strings to be numbers -- assuming␣
 ↪the character is a number
```

```
[1] "Arithmetic operators:"
```

4
-2
18
6

```
[1] "Power operations"
```

8
8
9
9

```
Warning message in sqrt(-1):
```

NaN
Inf
-Inf

```
[1] "Modular arithmetic"
```

1.5
1
2
1. 12 2. 4 3. 16 4. -18
1. 6 2. 4 3. 32 4. -54
'Hello World'
'HelloWorld'
'Hello World '
1. '6 Potato' 2. '2 Potato' 3. '8 Potato' 4. '-9 Potato'

[6]:
```
#Conditionals

#equality

1==1
1==2
1!=1
1!=2

#comparative
1>2
1<2
1>1
1>=1
2<=2
2<2


#Atomization
1 <= newVector #c(6, 2, 8, -9)

#N.B. TRUE and FALSE have numerical values as well!
#TRUE == 1 and FALSE == 0, so we can do something like this!

sum(newVector %% 2 == 0)

#strings
"Hello" == "hello" #Capitalization matters!
```

TRUE
FALSE
FALSE
TRUE
FALSE
TRUE
FALSE
TRUE
TRUE
FALSE

1. TRUE 2. TRUE 3. TRUE 4. FALSE

3

FALSE

[7]:
```
#Conditions!

#if-else
#If statements ask a question and execute subsequent code if that condition is␣
 ↪true

testVal <- 4

if(testVal == 4) {
    print("Yuh-yeet!")
}

if(testVal == 9) {
    print("Well, that's unexpected")
} else if(testVal == 15) {
    print("still quite weird")
} else {
    print("we finally made it!")
}

# neat function: ifelse

ifelse(testVal == 3, "if the condition is true", "Si la condicion es falsa")

testVec <- c(1,2,3,4)

#vectorization!
ifelse(testVec == 3, "si la condicion es verdadera", "if the condition is␣
 ↪false")
```

```
[1] "Yuh-yeet!"
[1] "we finally made it!"
```

'Si la condicion es falsa'

1. 'if the condition is false' 2. 'if the condition is false' 3. 'si la condicion es verdadera' 4. 'if the condition is false'

[8]:
```
#the real power of coding!
#Looping

#Basic construct


#for(index in iterable) {
#    do code
#}
```

```
for(index in 1:10) {
    print(index)
    if(index == 7) {
        print("We can do many things within this block!")
    }
    print(index*index)
}
```

```
[1] 1
[1] 1
[1] 2
[1] 4
[1] 3
[1] 9
[1] 4
[1] 16
[1] 5
[1] 25
[1] 6
[1] 36
[1] 7
[1] "We can do many things within this block!"
[1] 49
[1] 8
[1] 64
[1] 9
[1] 81
[1] 10
[1] 100
```

[9]:
```
#an iterable can be anything that can be iterated over
#basically 1:10 is equal to c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

for(index in newVector) {
    print(index)
}
```

```
[1] 6
[1] 2
[1] 8
[1] -9
```

[10]:
```
#Basic R functions

#getting attributes of your objects
```

```
length(newVector)
nrow(newMatrix)
ncol(newMatrix)
dim(newMatrix)
dim(dataframe)
length(dataframe) #N.B. gives the number of columns
dimnames(dataframe)
colnames(dataframe)

#strings
length("Hello")
nchar("Hello")
substr("Hello", 2, 4)

#Can use regular expressions with strsplit, grep, etc but won't talk about that␣
 ↪here
```

4
2
4
1. 2 2. 4
1. 4 2. 3
3

1. (a) ’1’ (b) ’2’ (c) ’3’ (d) ’4’
2. (a) ’col1’ (b) ’col2’ (c) ’col3’

1. ’col1’ 2. ’col2’ 3. ’col3’
1
5
’ell’

[11]:
```
#More advanced data-wrangling

#R Doesn't like for loops that much

#Apply family:

#Let's generate 10 dataframes in a list and do t-tests over them

baseName <- "df"

df.list <- list()

for(i in 1:10) {
    #df.list[x] names the dataframe
    df.list[[paste0(baseName, i)]] <- data.frame(
        #rnorm is a function that will generate n numbers from a normal␣
 ↪distribution with
```

```
        #mean and sd -- these are the 1st, 2nd, and 3rd args respectively.
        #runif is a similar function but using a uniform distribution from min
 ↪to max
        #n, min, and max are again the 1st, 2nd, and 3rd args respectively.
        x1 = rnorm(50, runif(1, 0, 5), runif(1, 0, 3)),
        x2 = rnorm(50, runif(1, 0, 5), runif(1, 0, 3))
    )
}


#generally, you'd think you'd just want to do something like this:
ttest.res <- list()
for(i in 1:length(df.list)){
    ttest.res[[paste0(baseName, i)]] <- t.test(df.list[[i]]$x1, df.
 ↪list[[i]]$x2)$p.value
}
```

[12]:
```
ttest.res
```

$df1 0.0785457394051292

$df2 3.63110402809643e-24

$df3 8.76578849232939e-14

$df4 0.206829196739598

$df5 2.04328230769023e-11

$df6 0.00156542257177538

$df7 0.00193441762636424

$df8 4.10998041918686e-22

$df9 1.05897638462207e-22

$df10 8.94922421122748e-17

[13]:
```
#lapply

lapply(df.list, function(x) t.test(x$x1, x$x2)$p.value)
```

$df1 0.0785457394051292

$df2 3.63110402809643e-24

$df3 8.76578849232939e-14

$df4 0.206829196739598

$df5 2.04328230769023e-11

$df6 0.00156542257177538

$df7 0.00193441762636424

$df8 4.10998041918686e-22

$df9 1.05897638462207e-22

$df10 8.94922421122748e-17

[14]:
```
#sapply

sapply(df.list, function(x) t.test(x$x1, x$x2)$p.value)


t.tests <- lapply(df.list, function(x) t.test(x$x1, x$x2))

p.val.results <- sapply(t.tests, function(x) x$p.value)

p.val.results

t.tests
```

df1 0.0785457394051292 **df2** 3.63110402809643e-24 **df3** 8.76578849232939e-14 **df4** 0.206829196739598 **df5** 2.04328230769023e-11 **df6** 0.00156542257177538 **df7** 0.00193441762636424 **df8** 4.10998041918686e-22 **df9** 1.05897638462207e-22 **df10** 8.94922421122748e-17
df1 0.0785457394051292 **df2** 3.63110402809643e-24 **df3** 8.76578849232939e-14 **df4** 0.206829196739598 **df5** 2.04328230769023e-11 **df6** 0.00156542257177538 **df7** 0.00193441762636424 **df8** 4.10998041918686e-22 **df9** 1.05897638462207e-22 **df10** 8.94922421122748e-17

```
$df1

Welch Two Sample t-test

data:  x$x1 and x$x2
t = -1.789, df = 61.431, p-value = 0.07855
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1.07268068  0.05955967
sample estimates:
mean of x mean of y
 1.707615  2.214175


$df2

Welch Two Sample t-test
```

```
data:  x$x1 and x$x2
t = 14.466, df = 81.043, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 2.983626 3.935233
sample estimates:
mean of x mean of y
 4.771056  1.311626


$df3

Welch Two Sample t-test

data:  x$x1 and x$x2
t = 8.9055, df = 84.521, p-value = 8.766e-14
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 2.752762 4.335424
sample estimates:
mean of x mean of y
 4.670857  1.126765


$df4

Welch Two Sample t-test

data:  x$x1 and x$x2
t = -1.2734, df = 74.737, p-value = 0.2068
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1.1599758  0.2553383
sample estimates:
mean of x mean of y
 4.120103  4.572422


$df5

Welch Two Sample t-test

data:  x$x1 and x$x2
t = -7.6906, df = 87.432, p-value = 2.043e-11
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1.4257905 -0.8401946
sample estimates:
```

```
mean of x mean of y
 3.499264  4.632256


$df6

Welch Two Sample t-test

data:  x$x1 and x$x2
t = -3.2704, df = 82.895, p-value = 0.001565
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.4145417 -0.5882921
sample estimates:
mean of x mean of y
 2.230623  3.732040


$df7

Welch Two Sample t-test

data:  x$x1 and x$x2
t = 3.2474, df = 58.257, p-value = 0.001934
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.4206395 1.7722241
sample estimates:
mean of x mean of y
 4.191411  3.094979


$df8

Welch Two Sample t-test

data:  x$x1 and x$x2
t = -12.886, df = 89.627, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -4.095236 -3.001127
sample estimates:
 mean of x  mean of y
-0.0024916  3.5456901


$df9
```

```
Welch Two Sample t-test

data:  x$x1 and x$x2
t = 13.947, df = 76.426, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 3.642341 4.855761
sample estimates:
  mean of x   mean of y
 4.18019906 -0.06885193


$df10

Welch Two Sample t-test

data:  x$x1 and x$x2
t = -10.109, df = 95.728, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -2.512752 -1.687868
sample estimates:
mean of x mean of y
  2.43885   4.53916
```

[15]:
```
#apply

single.df <- df.list[[1]]

apply(single.df, 1, mean)

apply(single.df, 2, mean)
```

1. 2.31091743235557 2. 2.85485229844394 3. 1.18795815664118 4. 3.54321387777942 5. 0.473395091627835 6. 2.39539467531031 7. 3.01342400558742 8. 0.819699290815774 9. 1.14906426803454 10. 2.21550315876291 11. 0.320045692605368 12. 2.26910603542297 13. 0.87843307616928 14. 2.22343346684886 15. 2.12113641153013 16. 1.47660255097294 17. 3.51789199395194 18. 2.23623331833236 19. 2.34654949891397 20. 2.73690666405483 21. 1.62609248857357 22. 3.07176736935125 23. 2.00102182785066 24. 2.73589805526344 25. 1.00555190171365 26. 2.79227354460853 27. 0.0207163497380278 28. 1.31039683812737 29. 2.00159985003325 30. 3.13199748079402 31. 0.207097126797536 32. 2.82993220436128 33. 2.26020276450515 34. 3.38762451598377 35. 3.68539033695579 36. 1.23069743321836 37. 1.94252071917726 38. 1.83254057825486 39. 2.85207081508311 40. 1.05031566507531 41. 1.25806878681484 42. 2.58439654921881 43. 2.01435690661913 44. 1.56560754346955 45. -0.056422466104329 46. 1.32251981039774 47. 3.43368744190514 48. 0.951063535369841 49. 1.53889966085235 50. 2.36711259061371

x1 1.70761493185855 **x2** 2.21417543569283

[16]:
```r
#reading data

#In RStudio, you can do this interactively with "Import Dataset" in the top␣
 ↪right of the window
#basic form: read.`type`(path, options)
ratData <- read.table(
    "./dataFiles/rat_KD.txt",
    header=T,
    row.names=1
)
```

[17]:
```r
#basic statistical tests and data transformations

#How many samples are there?
ncol(ratData)

#How many probes per sample?
nrow(ratData)

#Mean for sample 1?
mean(ratData[, 1])
#Mean for all samples individually?
apply(ratData, 2, mean)
#for each probe?
apply(ratData[1:10, ], 1, mean)

#Let's transform the data
ratData.log <- log2(ratData)

head(ratData.log)


#median will report the median
#max or min, will report the max/min respecitvely

#Let's do a t-test on the first 10 probes and get the p-values

ratData.log.sub <- ratData.log[1:10, ] #rows 1 to 10, all columns
#columns 1:6 are controls, columns 7:11 are the treatment

ratData.ttest.sub <- apply(ratData.log.sub, 1, function(x) t.test(x[1:6], x[7:
 ↪11], var.equal = T))

sapply(ratData.ttest.sub, function(x) signif(x$p.value, 3))

#what if we didn't want to assume normality?
```

```
ratData.mwu.sub <- apply(ratData.log.sub, 1, function(x) wilcox.test(x[1:6],␣
↪x[7:11]))
sapply(ratData.mwu.sub, function(x) signif(x$p.value, 3))

#count data:

#Let's make a simulated matrix:

fisher.test(
    matrix(
        round(runif(4, 10, 35), 0),
        nrow = 2
    )
)



#other tests: aov (ANOVA), cor.test (linear correlation), ks.test␣
↪(Kolmogorov-Smirnov), chisq.test(), etc,

#lm() is for linear modeling, also not covered here
```

11

15923

180.237430285455

control.diet.19300 180.237430285455 **control.diet.19301** 180.171667462846 **control.diet.19302** 180.0602907675 **control.diet.19303** 180.851004878044 **control.diet.19304** 179.968331882453 **control.diet.19305** 184.077788248264 **ketogenic.diet.19306** 183.388014190875 **ketogenic.diet.19307** 185.416908450173 **ketogenic.diet.19308** 186.541537359643 **ketogenic.diet.19309** 184.216884312485 **ketogenic.diet.19310** 182.199126855517

AFFX-BioB-5_at 88.6301181818182 **AFFX-BioB-M_at** 108.792018181818 **AFFX-BioB-3_at** 45.3780454545455 **AFFX-BioC-5_at** 189.516090909091 **AFFX-BioC-3_at** 113.0306 **AFFX-BioDn-5_at** 164.402181818182 **AFFX-BioDn-3_at** 1187.43363636364 **AFFX-CreX-5_at** 1970.96090909091 **AFFX-CreX-3_at** 2738.66 **AFFX-DapX-5_at** 11.2185909090909

A data.frame: 6 Œ 11

	control.diet.19300 <dbl>	control.diet.19301 <dbl>	control.diet.19302 <dbl>	contr <dbl>
AFFX-BioB-5_at	6.256301	6.427351	6.333542	6.552
AFFX-BioB-M_at	6.565442	6.198643	6.470522	6.801
AFFX-BioB-3_at	4.801340	5.477292	5.082464	5.907
AFFX-BioC-5_at	7.445752	7.246256	7.387406	7.647
AFFX-BioC-3_at	6.443416	6.555118	6.910289	6.840
AFFX-BioDn-5_at	7.229953	7.081798	7.352123	7.417

AFFX-BioB-5_at 0.32 **AFFX-BioB-M_at** 0.00304 **AFFX-BioB-3_at** 0.195 **AFFX-BioC-5_at** 0.0374 **AFFX-BioC-3_at** 0.0176 **AFFX-BioDn-5_at** 0.0401 **AFFX-BioDn-3_at** 0.494 **AFFX-CreX-5_at** 0.0983 **AFFX-CreX-3_at** 0.814 **AFFX-DapX-5_at** 0.0178

AFFX-BioB-5_at 0.329 **AFFX-BioB-M_at** 0.00433 **AFFX-BioB-3_at** 0.247 **AFFX-BioC-5_at** 0.0519 **AFFX-BioC-3_at** 0.0303 **AFFX-BioDn-5_at** 0.0303 **AFFX-BioDn-3_at** 0.931 **AFFX-CreX-5_at** 0.177 **AFFX-CreX-3_at** 0.662 **AFFX-DapX-5_at** 0.00433

```
Fisher's Exact Test for Count Data

data:  matrix(round(runif(4, 10, 35), 0), nrow = 2)
p-value = 0.677
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.3095338 1.8863586
sample estimates:
odds ratio
 0.7706811
```

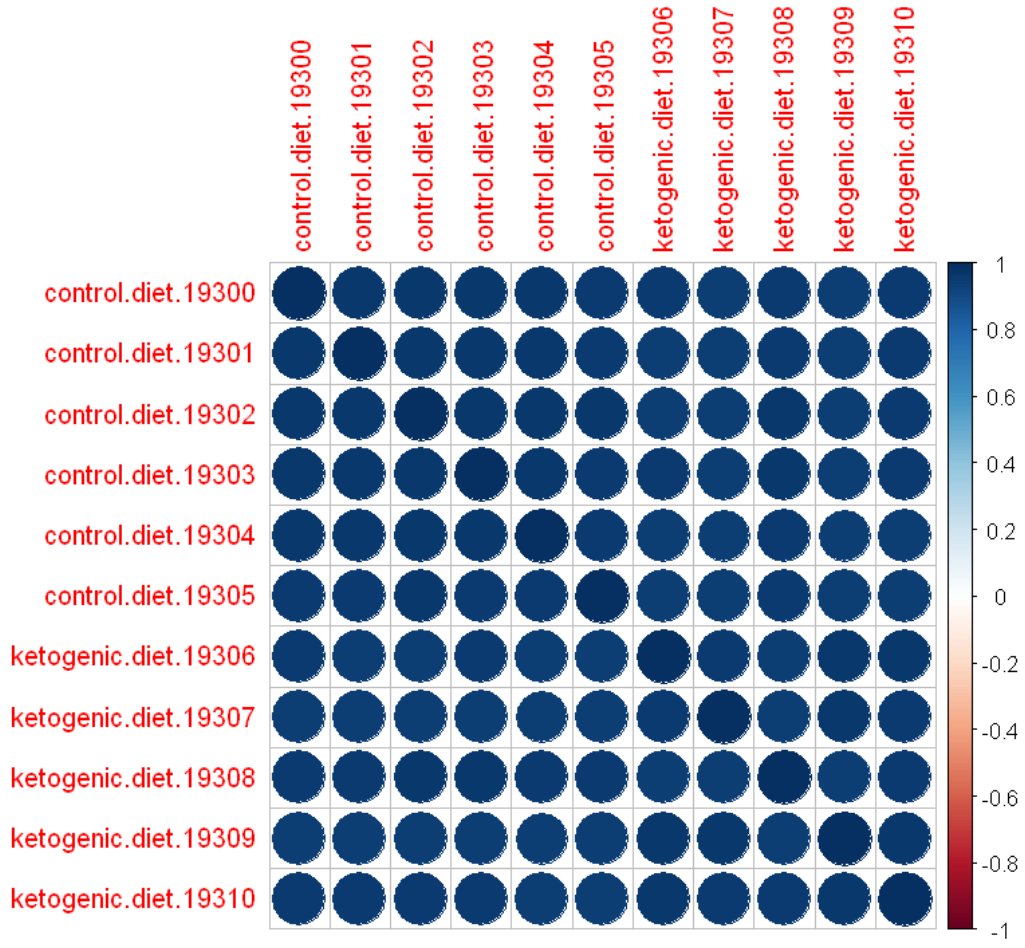
[18]:
```r
#installing packages

#Sometimes you want to use non-base functions

#e.g. let's install the library corrplot to make a correllogram

install.packages('corrplot') #installs the package into your profile
#Common useful packages:
#ggplot2, ggpubr, dplyr, magrittr, readxl, knitr

#Bioconductor:
#this is a suite of packages that are bioinformatics related and somewhat␣
 ↪tricky to manage, but very useful
#The bioconductor documentation is very very good usually
#install by:
#install.packages("BiocManager")
#install a bioconductor package with:
#BiocManager::install(<package>)
#e.g. I use ChIPpeakAnno a lot for integration site analyses
#BiocManager::install("ChIPpeakAnno")
#library(ChIPpeakAnno)
#There are also dataset packages that have listings for the human genome
#e.g. TxDb.Hsapiens.UCSC.hg19.knownGene

#The main function we care about is corrplot; if we just call it like this:

corrplot(cor(ratData.log), title="Correlation Plot")
```

```
Installing package into 'C:/Users/Michael/Documents/R/win-library/3.6'
(as 'lib' is unspecified)

package 'corrplot' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Michael\AppData\Local\Temp\Rtmp25gkVV\downloaded_packages
```

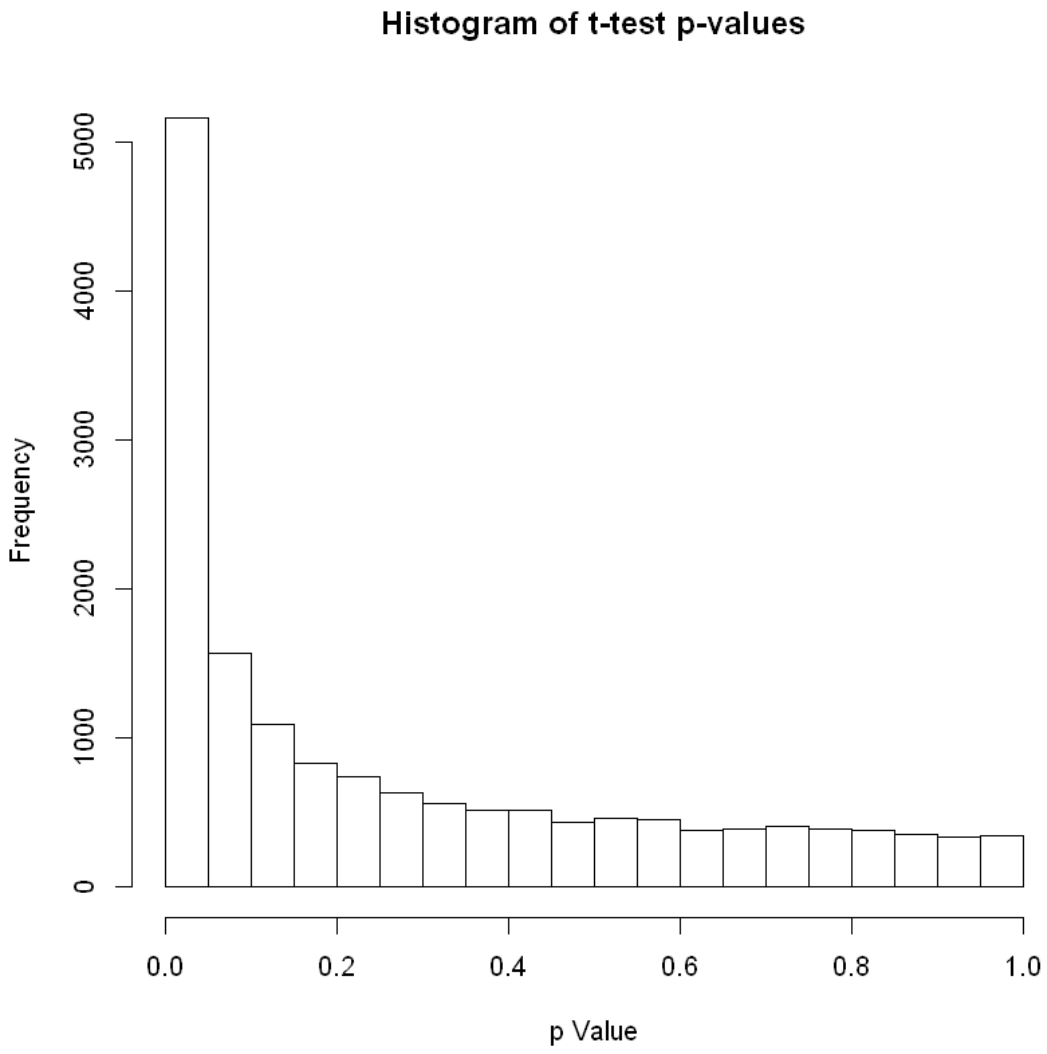
```
        Error in corrplot(cor(ratData.log), title = "Correlation Plot"): could␣
↪not find function "corrplot"
    Traceback:
```

[19]:
```
library(corrplot) #makes the library functions available

corrplot(cor(ratData.log))
```

```
Warning message:
"package 'corrplot' was built under R version 3.6.3"corrplot 0.84 loaded
```



[20]:
```
#practical

#Okay, let's go through a whole set of analyses with the rat data

#We're going to repeat a couple of steps, but that's okay

#Read in the data from the file:
ratData <- read.table("./dataFiles/rat_KD.txt", header=T, row.names=1)

#log2 transform:
ratData.log <- log2(ratData)

#Let's perform t-tests on all of the probes and observe the distribution of␣
 ↪p-values
#This will also serve as a brief introduction to plotting in base-R
#There is another jupyter notebook on the GH repo that talks about using ggplot2

#This will take a second to run
ratData.p <- apply(ratData.log, 1, function(x) t.test(x[1:6], x[7:11], var.
 ↪equal=T)$p.value)

#Observe distribution of p-values via a histogram

hist(
    ratData.p,
    main="Histogram of t-test p-values",
    xlab="p Value"
)
```



[21]:
```
#Okay, there are a lot of potentially siginificant p-values
#how many are below the standard 0.05 and 0.01

#remember, TRUE and FALSE have numerical values!

sum(ratData.p <0.05)
sum(ratData.p <0.01)

#we can also adjust p-values for multiple corrections

#Valid methods -- you can google the differences if you so choose
p.adjust.methods
```

```
sum(p.adjust(ratData.p, "bonferroni") < 0.01)
```

5160

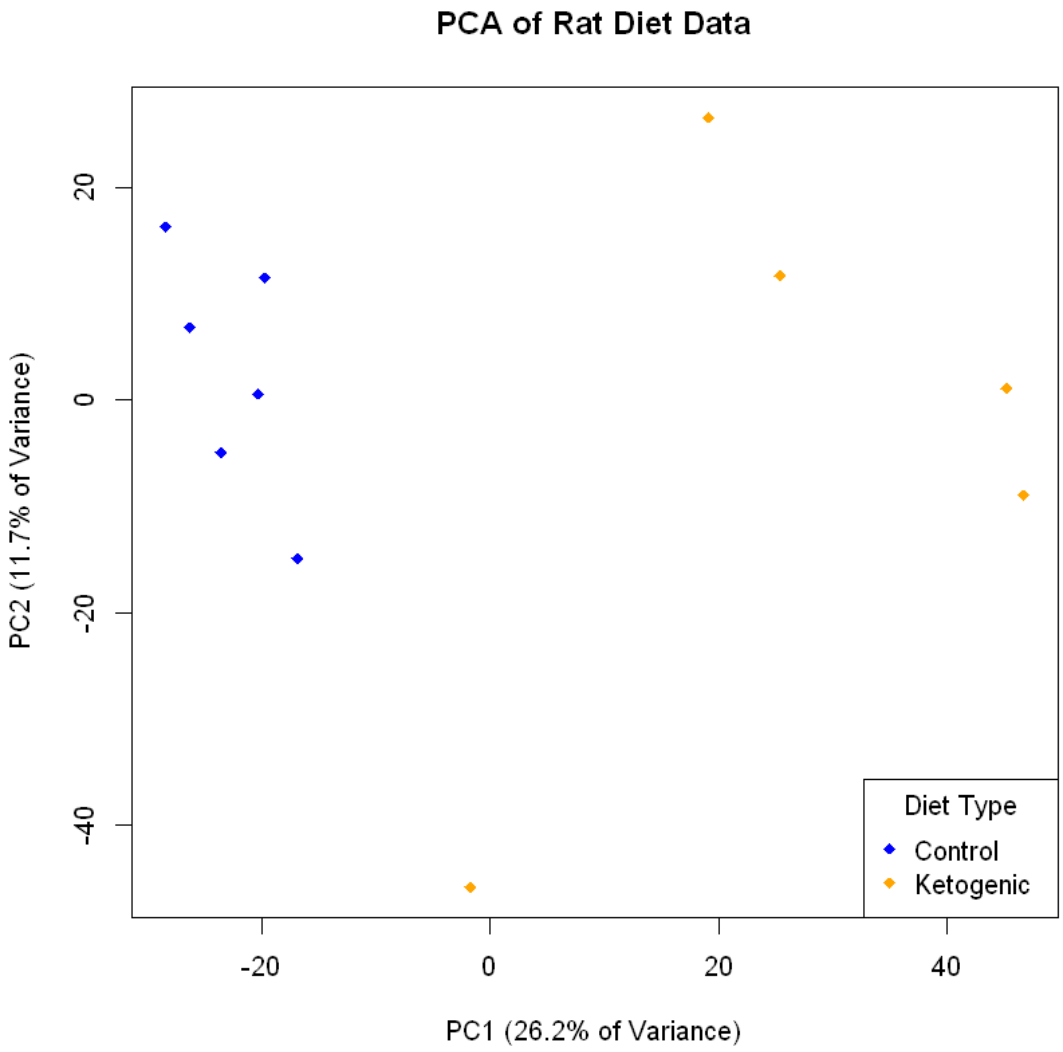
2414

1. 'holm' 2. 'hochberg' 3. 'hommel' 4. 'bonferroni' 5. 'BH' 6. 'BY' 7. 'fdr' 8. 'none'

8

[22]:
```
#Let's visualize the data using dimensionality reduction

#DR is basically taking high-dimensional data, and plotting in an x-y plane (or␣
 ↪x-y-z)
#(remember, rat data has ~16k data points/sample you can't visualize a 16k␣
 ↪dimensional graph)

#Let's set up a couple of variables that will be useful

#sample type
sample_type <- c(rep(1, 6), rep(2, 5)) #rep(x, n) is a function that replicates␣
 ↪x, n times
colors <- c("blue", "orange")


#Perform principal component analysis

#This takes the n-dimensional data and projects it
#t() is the transpose function
#the transpose of a matrix is the matrix formed by the reversing the row-column␣
 ↪indexes
#e.g. mat[i,j] == t(mat[j, i])
ratData.pca <- prcomp(t(ratData.log))

#It is standard form to have the percent variance noted with each Principal␣
 ↪compenent, so:
pc1.var <- ratData.pca$sdev[1]^2 / sum(ratData.pca$sdev^2)
#Remember, since we get a vector back from ratData.pca$sdev, performing ^2 on␣
 ↪it squares every element
#sdev is a vector of the standard deviations; we want the variances which is␣
 ↪the square of sdev
#plotting in two-dimensions so
pc2.var <- ratData.pca$sdev[2]^2 /sum(ratData.pca$sdev^2)

#okay, let's make a plot!
#basic plot function to create the outline:

plot(
    #first two lines creates the axis ranges
    x = range(ratData.pca$x[, 1]),
```

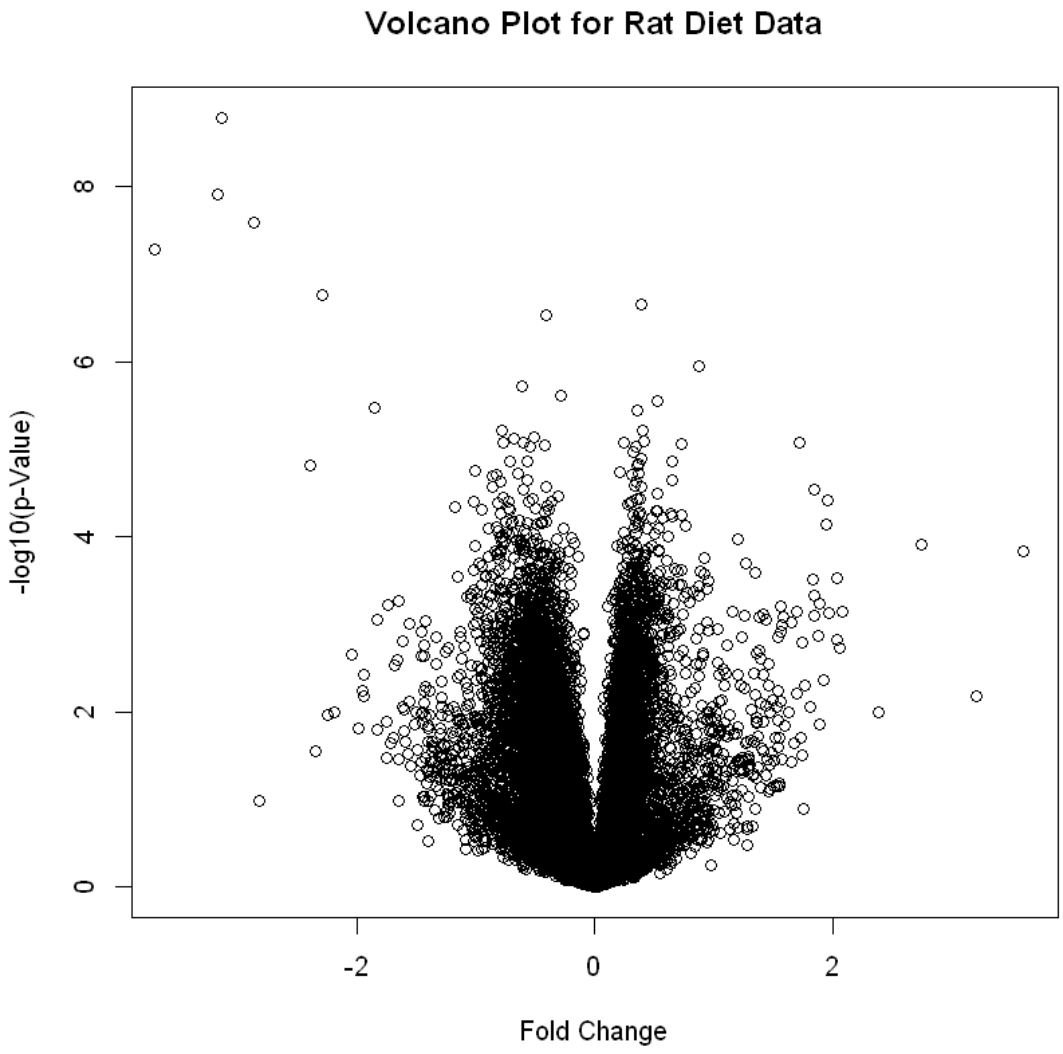
```
    y = range(ratData.pca$x[, 2]),
    #Don't plot the range points, just make the axes
    type='n',
    #Figure labels
    #Title
    main = "PCA of Rat Diet Data",
    #X-axis label
    xlab = paste0(
        "PC1 (",
        round(pc1.var * 100, 1),
        "% of Variance)"
    ),
    #y-axis label
    ylab = paste0(
        "PC2 (",
        round(pc2.var * 100, 1),
        "% of Variance)"
    )
)
#Add the data points
points(
    #Data points
    x = ratData.pca$x[, 1],
    y = ratData.pca$x[, 2],
    #Color them by type
    #colors is blue or pink, sample type is 1 or 2 so
    col = colors[sample_type],
    #pch -- point character
    pch = 18
)
#Add a legend
legend(
    "bottomright",
    title="Diet Type",
    legend=c("Control", "Ketogenic"),
    col=colors,
    pch=18
)
```



[23]:
```
#Clearly the data are segregated, so let's look at those significant p-values␣
 ↪we generated before
#What effect on expression

#Let's make a volcano plot comparing log-transformed p-values to the fold␣
 ↪change in expression

#First, let's transform the p-values

ratData.Logp <- (-1)*log10(ratData.p) #Gives the order of magnitude of the␣
 ↪p-values

#Now, let's get the fold change
```

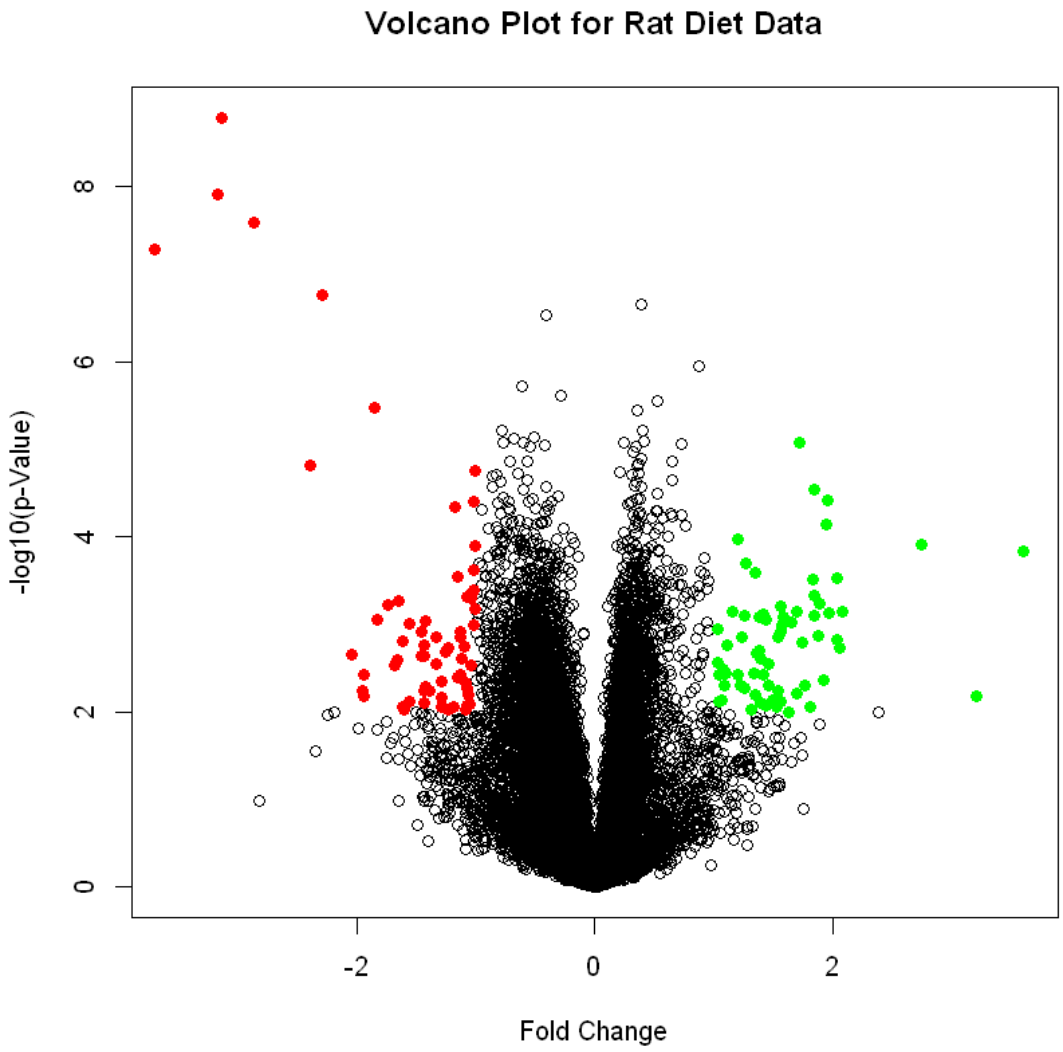
```
#Remember, we're in log-space, so it's just the difference in the means

ratData.control.mean <- apply(ratData.log[, 1:6], 1, mean)
ratData.ketogenic.mean <- apply(ratData.log[, 7:11], 1, mean)

ratData.foldChange <- ratData.ketogenic.mean - ratData.control.mean

#Okay, we have all we need!
```

[24]:
```
#First step, generate the axes:

plot(
    #Make the plot symmetrical about 0
    #x-range goes from the negative absolute value of max fold change to␣
 ↪positive value
    x=range(
        c(
            -abs(max(ratData.foldChange)),
            abs(max(ratData.foldChange))
        )
    ),
    #y-range will be from 0 to the max value in the log-transformed p-values
    y=range(
        c(0, max(ratData.Logp))
    ),
    #don't show these points
    type='n',
    #Labels
    #Title
    main="Volcano Plot for Rat Diet Data",
    #x-axis
    xlab="Fold Change",
    #y-axis
    ylab="-log10(p-Value)"
)
#Add points
points(
    x=ratData.foldChange,
    y=ratData.Logp,
    col='black',
    pch=21
)
```



[25]:
```
#Okay, that's only somewhat helpful
#Let's color the values that have a p-value that is significant at the 99%␣
 ↪confidence level
#Also, let's divide those colors into up-regulate (FC > 1) and down-regulate␣
 ↪(FC < -1)

plot(
    #Make the plot symmetrical about 0
    #x-range goes from the negative absolute value of max fold change to␣
 ↪positive value
    x=range(
        c(
            -abs(max(ratData.foldChange)),
```

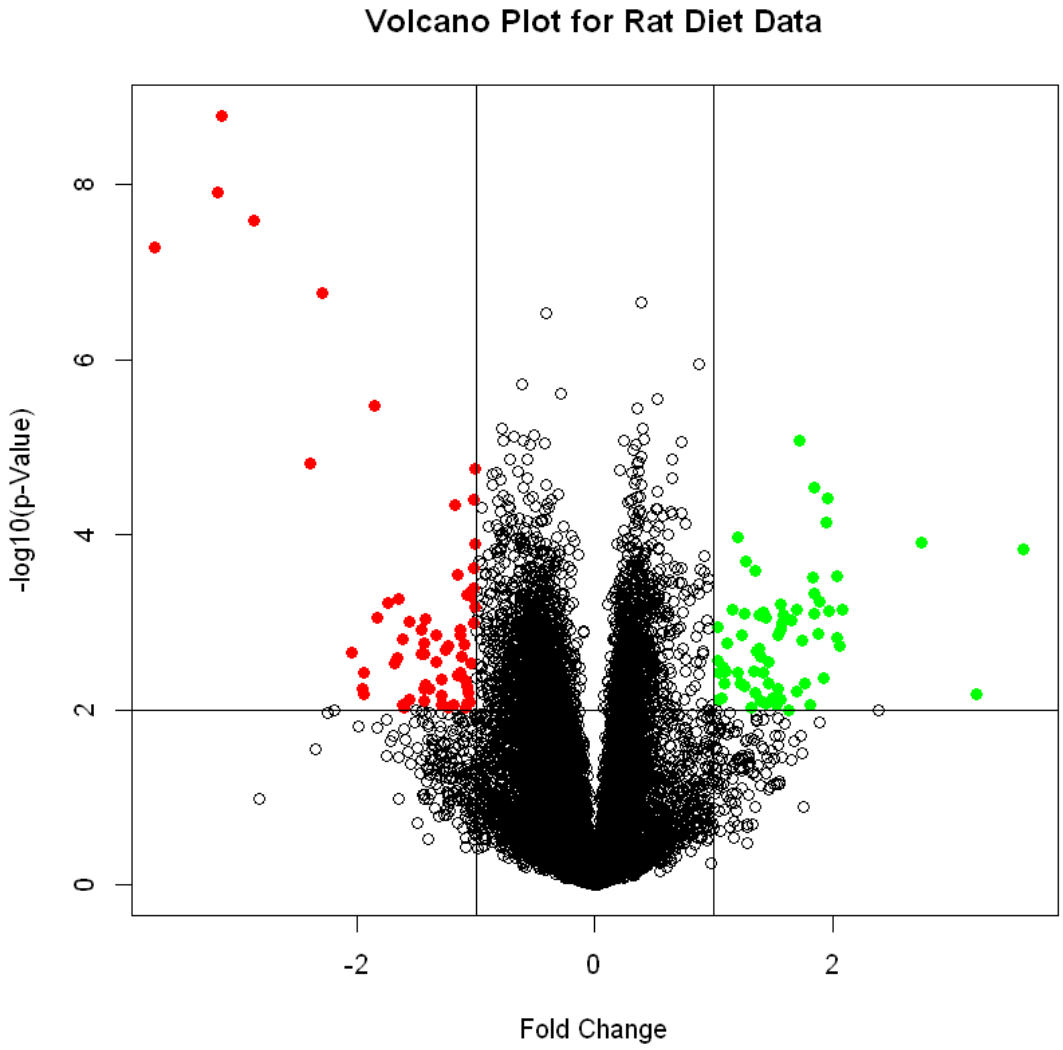
```
            abs(max(ratData.foldChange))
        )
    ),
    #y-range will be from 0 to the max value in the log-transformed p-values
    y=range(
        c(0, max(ratData.Logp))
    ),
    #don't show these points
    type='n',
    #Labels
    #Title
    main="Volcano Plot for Rat Diet Data",
    #x-axis
    xlab="Fold Change",
    #y-axis
    ylab="-log10(p-Value)"
)
#Add points
points(
    x=ratData.foldChange,
    y=ratData.Logp,
    col='black',
    pch=21
)
#Color the points of interest
points(
    #Only the data points with significant p-values and up-regulation
    x = ratData.foldChange[
        ratData.foldChange > 1 & ratData.Logp > -log10(0.01)
    ],
    y = ratData.Logp[
        ratData.foldChange > 1 & ratData.Logp > -log10(0.01)
    ],
    pch=19,
    col="green"
)
points(
    #Only the data points with significant p-values and down-regulation
    x = ratData.foldChange[
        ratData.foldChange < -1 & ratData.Logp > -log10(0.01)
    ],
    y = ratData.Logp[
        ratData.foldChange < -1 & ratData.Logp > -log10(0.01)
    ],
    pch=19,
    col="red"
)
```



[26]:
```
#One last thing:
#Let's add lines to show the sextants

#old plot from previous cell

plot(
    #Make the plot symmetrical about 0
    #x-range goes from the negative absolute value of max fold change to␣
 ↪positive value
    x=range(
        c(
            -abs(max(ratData.foldChange)),
            abs(max(ratData.foldChange))
```

```
        )
    ),
    #y-range will be from 0 to the max value in the log-transformed p-values
    y=range(
        c(0, max(ratData.Logp))
    ),
    #don't show these points
    type='n',
    #Labels
    #Title
    main="Volcano Plot for Rat Diet Data",
    #x-axis
    xlab="Fold Change",
    #y-axis
    ylab="-log10(p-Value)"
)
#Add points
points(
    x=ratData.foldChange,
    y=ratData.Logp,
    col='black',
    pch=21
)
#Color the points of interest
points(
    #Only the data points with significant p-values and up-regulation
    x = ratData.foldChange[
        ratData.foldChange > 1 & ratData.Logp > -log10(0.01)
    ],
    y = ratData.Logp[
        ratData.foldChange > 1 & ratData.Logp > -log10(0.01)
    ],
    pch=19,
    col="green"
)
points(
    #Only the data points with significant p-values and down-regulation
    x = ratData.foldChange[
        ratData.foldChange < -1 & ratData.Logp > -log10(0.01)
    ],
    y = ratData.Logp[
        ratData.foldChange < -1 & ratData.Logp > -log10(0.01)
    ],
    pch=19,
    col="red"
)
#abline is a function that just draws lines
```

```
#let's add two vertical lines and a horizontal line
abline(h = -log10(0.01))
abline(v = -log2(2))
abline(v = log2(2))

#The h argument just draws a horizontal line acress the plot at y = h
#Conversely, v does the same with a vertical line at v = x
#Alternatively, draw a sloped line of form y = a + b*x with abline(a, b)
```



[ ]: