Data Analytics II/III

QBS 103: Foundations of Data Science

August 6, 2024

Lesson Objectives

At the end of this lecture you should be able to:

1. Calculate appropriate summary statistics for a data frame
2. Apply a single function across multiple rows/columns
3. Build a simple correlation plot
4. Build a heatmap

Additional Resources

corrplot: https://cran.r-project.org/web/packages/corrplot/vignettes/corrplot-intro.html

pheatmap: https://r-charts.com/correlation/pheatmap/

Identifying Appropriate Summary Statistics

First, lets generate some random data.

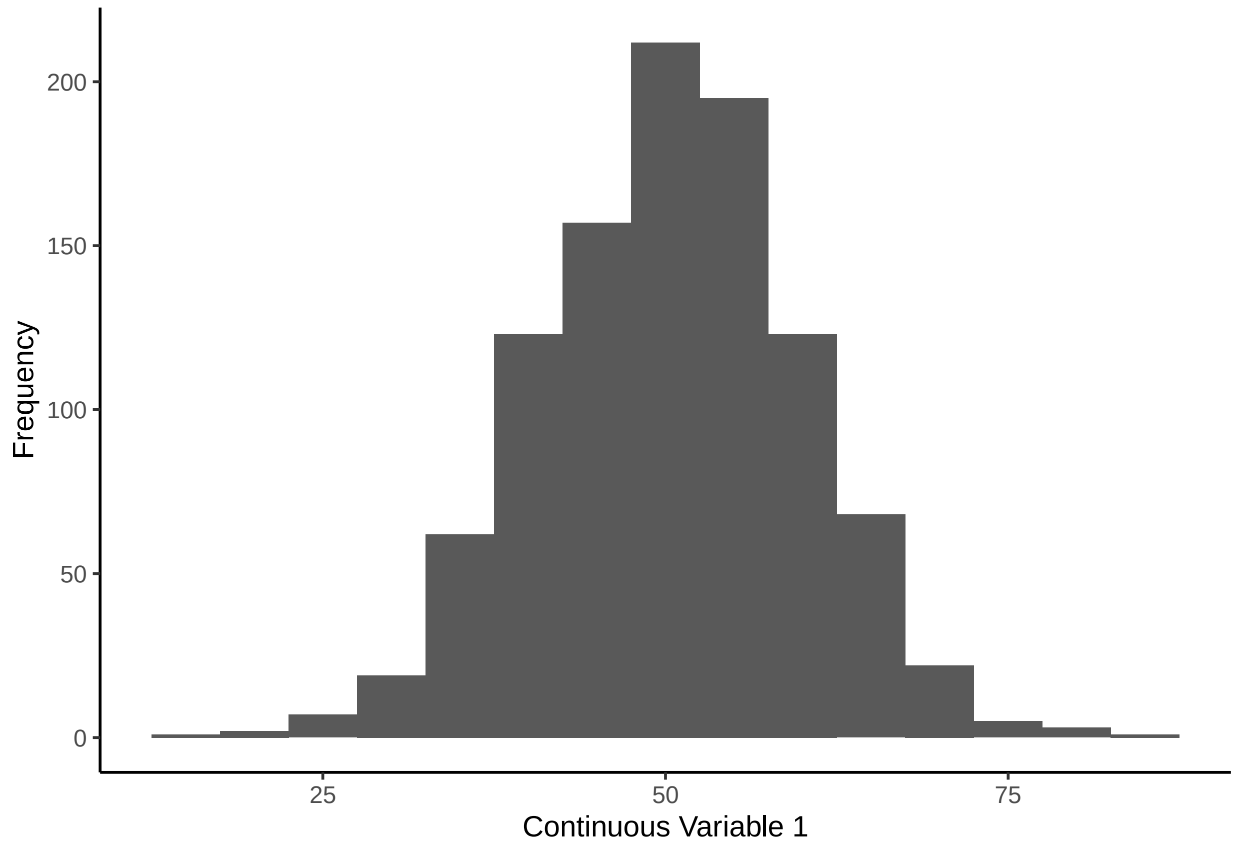
```r
library(tidyverse)
set.seed(3927)

randomData <- data.frame(Cont1 = rnorm(n = 1000,mean = 50,sd = 10),
                         Cont2 = rnorm(n = 1000, mean = 23,sd = 2),
                         Cont3 = rexp(n = 1000,rate = 0.25),
                         Cat1 = factor(rbinom(n = 1000,size = 1,prob = 0.50),
                                       labels = c(F,T)),
                         Cat2 = factor(rbinom(n = 1000,size = 1,prob = 0.25),
                                       labels = c(F,T)),
                         Cat3 = factor(rbinom(n = 1000,size = 2,prob = 0.40),
                                       labels = c('A','B','C')))
```

Continuous Variables

For normally distributed continuous variables, we report mean and standard deviation.

```r
ggplot(data = randomData,aes(x = Cont1)) +
  geom_histogram(binwidth = 5) +
  labs(x = 'Continuous Variable 1',y = 'Frequency') +
  theme_classic()
```



We can extract all the individual components of mean and standard deviation using the following functions.

```
# Mean
mean(randomData$Cont1)
```

```
## [1] 50.34092
```

```
# Standard deviation
sd(randomData$Cont3)
```

```
## [1] 3.76435
```

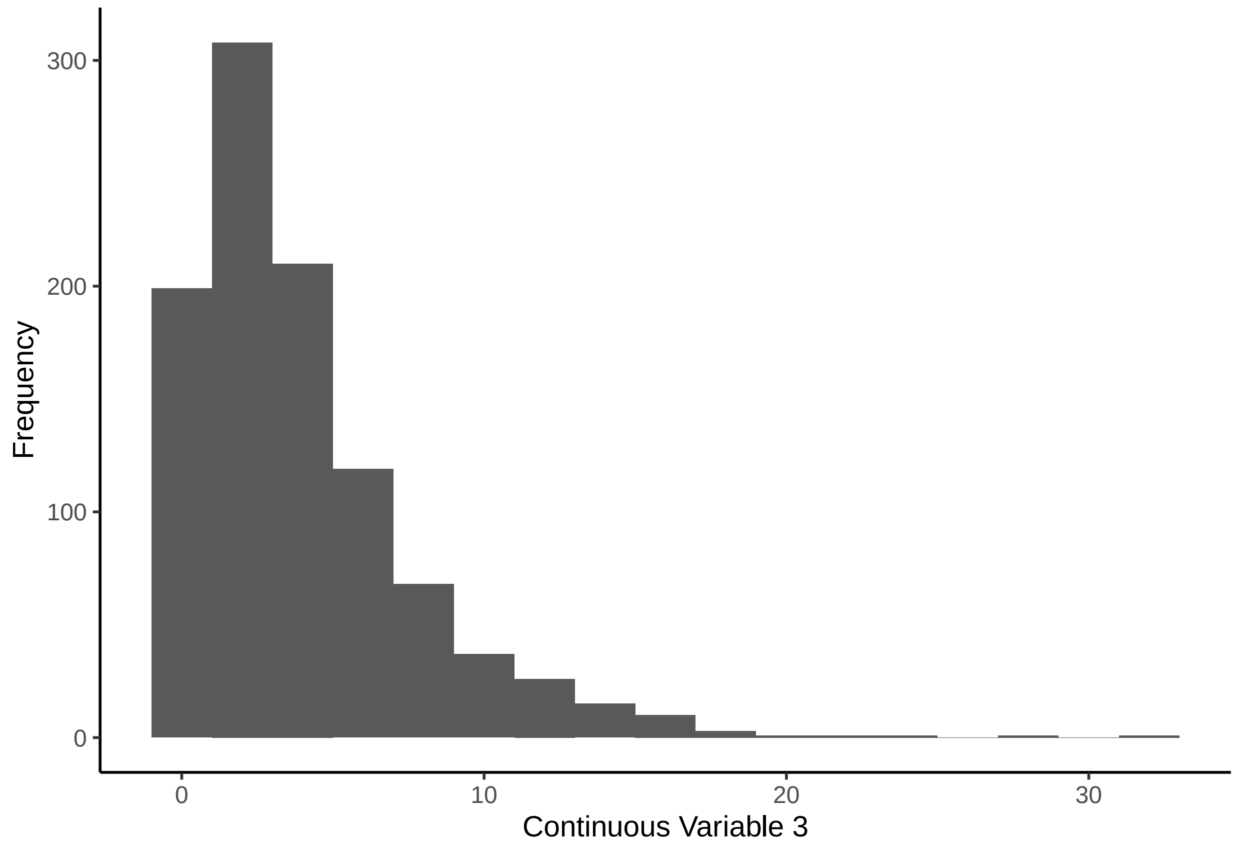
We can then combine these values to make them nicely formatted.

```
print(paste0('Mean (sd): ',round(mean(randomData$Cont1),digits = 2),' (',
             round(sd(randomData$Cont1),digits = 2),')'))
```

```
## [1] "Mean (sd): 50.34 (9.61)"
```

For non-normally distributed continuous variables, we report median and interquartile range (IQR).

```
ggplot(data = randomData,aes(x = Cont3)) +
  geom_histogram(binwidth = 2) +
  labs(x = 'Continuous Variable 3',y = 'Frequency') +
  theme_classic()
```



We can extract all the individual components of median and IQR using the following functions.

```
# Median
median(randomData$Cont3)
```

```
## [1] 2.927949
```

```
# Quartile values
quantile(randomData$Cont3)
```

```
##           0%          25%          50%          75%         100%
##  0.001975371  1.293967997  2.927949040  5.380989123 31.038499748
```

```
# IQR (Q3 - Q1)
IQR(randomData$Cont3)
```

```
## [1] 4.087021
```

We can string these values together as follows to print out a clean and easy to read summary.

```
print(paste0('Median [IQR]: ',round(median(randomData$Cont3),digits = 2),' [',
             round(quantile(randomData$Cont3,1/4),digits = 2),', ',
             round(quantile(randomData$Cont3,3/4),digits = 2),']'))
```

```
## [1] "Median [IQR]: 2.93 [1.29, 5.38]"
```

Categorical Variables

For categorical covariates, we typically report the count (n) and percentage of each level of that variable.

```r
data.frame('n' = c(table(randomData$Cat1)),
           'perc' = c(round(table(randomData$Cat1)/1000*100,digits = 2)))
```

```
##         n perc
## FALSE 495 49.5
## TRUE  505 50.5
```

Refresher: Building a Function

To define a function in R, we use the following syntax:

```r
# Define function to calculate mean (sd)
meanSD <- function(x) {
  # Calculate individual values
  myMean <- mean(x)
  mySD <- sd(x)
  # Combine values
  paste0(round(myMean,digits = 2),' (',round(mySD,digits = 2),')')
}

meanSD(x = randomData$Cont1)
```

```
## [1] "50.34 (9.61)"
```

When we run a function, no intermediate values are saved. The only output from the function will be the final value you return.

We can also provide default values for terms in a function.

```r
# Define a function to calculate a mean or a median
contSummary <- function(x,normal = T) {

  # Calculate mean (sd) if normally distributed (the default)
  if (normal == T) {
      # Calculate individual values
    myMean <- round(mean(x),2)
    mySD <- round(sd(x),2)
    # Combine values
    paste0(myMean,' (',mySD,')')
  }
  # Calculate median (IQR) if non-normally distributed
  else {
    # Calculate individual values
    myMedian <- round(median(x))
    myIQR1 <- round(quantile(x,1/4),digits = 2)
    myIQR2 <- round(quantile(x,3/4),digits = 2)
    # Combine values
    paste0(myMedian,' [',myIQR1,', ',myIQR2,']')
  }
}

# Run function on normally distributed variable
contSummary(x = randomData$Cont1,normal = T)
```

```
## [1] "50.34 (9.61)"
```

```r
# Run function on non-normally distributed variable
contSummary(x = randomData$Cont3,normal = F)
```

```
## [1] "3 [1.29, 5.38]"
```

If we don't specify the "normal" term which we set a default for, the function will assume that it is normally distributed.

```r
contSummary(x = randomData$Cont1)
```

```
## [1] "50.34 (9.61)"
```

```r
contSummary(x = randomData$Cont3)
```

```
## [1] "3.98 (3.76)"
```

The *apply* Function

First, let's generate a new data set with 20 different continuous, normally distributed variables.

```r
set.seed(1234)
# Building empty data set
randomData2 <- as.data.frame(matrix(nrow = 100,ncol = 20))
# Generate 20 column names
colnames(randomData2) <- paste0('contVar',seq(1:ncol(randomData2)))
# Generate random means and SDs
means <- runif(20,min = 10,max = 30)
sds <- runif(20,min = 0.5,max = 5)
# Generate 20 normally distributed variables
for (i in 1:ncol(randomData2)) {
  randomData2[,i] <- rnorm(100,mean = means[i],sd = sds[i])
}
```

Say we wanted to look at the mean of each of these variables. We could loop through and calculate the mean iterativly for each variable like this:

```r
# Generate am empty vector of means
meansCalc1 <- array(dim = ncol(randomData2))
names(meansCalc1) <- colnames(randomData2)
# Look at our empty vector
meansCalc1
```

```
##  contVar1  contVar2  contVar3  contVar4  contVar5  contVar6  contVar7  contVar8
##        NA        NA        NA        NA        NA        NA        NA        NA
##  contVar9 contVar10 contVar11 contVar12 contVar13 contVar14 contVar15 contVar16
##        NA        NA        NA        NA        NA        NA        NA        NA
## contVar17 contVar18 contVar19 contVar20
##        NA        NA        NA        NA
```

```
# Loop through each column
for (i in colnames(randomData2)) {
  # Calculate the mean of values in that column
  meansCalc1[i] <- mean(randomData2[,i])
}

# Print final vector
meansCalc1
```

```
##  contVar1  contVar2  contVar3  contVar4  contVar5  contVar6  contVar7  contVar8
## 12.025310 22.648520 22.348987 22.402189 27.259461 22.323295  9.881249 14.690074
##  contVar9 contVar10 contVar11 contVar12 contVar13 contVar14 contVar15 contVar16
## 23.180793 20.238427 23.850068 20.950427 15.724085 28.933707 15.823120 26.325279
## contVar17 contVar18 contVar19 contVar20
## 15.512789 15.515943 14.436562 14.392330
```

```
# Compare to the list of means we used to generate our data
cbind(means,meansCalc1)
```

```
##              means meansCalc1
## contVar1  12.27407  12.025310
## contVar2  22.44599  22.648520
## contVar3  22.18549  22.348987
## contVar4  22.46759  22.402189
## contVar5  27.21831  27.259461
## contVar6  22.80621  22.323295
## contVar7  10.18992   9.881249
## contVar8  14.65101  14.690074
## contVar9  23.32168  23.180793
## contVar10 20.28502  20.238427
## contVar11 23.87183  23.850068
## contVar12 20.89950  20.950427
## contVar13 15.65467  15.724085
## contVar14 28.46867  28.933707
## contVar15 15.84632  15.823120
## contVar16 26.74591  26.325279
## contVar17 15.72447  15.512789
## contVar18 15.33642  15.515943
## contVar19 13.73446  14.436562
## contVar20 14.64452  14.392330
```

This requires several lines of code and, as we start working with bigger data, can be computationally intensive. Luckily, we can also do this with a single line of code using the *apply()* function. For this function, you will provide 3 inputs:

1. The data frame (m x n) you wan to use the apply function on. Note: The apply function will include every row and every column so ensure that you provide it with an appropriate subset of your data if necessary.
2. The "margin" i.e. if you want it to perform this function on rows or columns."**MARGIN = 1**" indicates it should perform the function on every on every row, resulting in an array of **m length** and "**MARGIN = 2**" indicates it should perform the function on every column, resulting in an array of **n length**.
3. The function you want it to apply on those rows or columns. You can either specify an existing function by name or define your own in line (more on this later).

So, we can recreate the same array of means we created in the loop above, now using a single line of code like this:

```
# Calculate the average of each row
meansCalc2 <- apply(randomData2,MARGIN = 2,FUN = mean)

# Compare to our previous calculations
cbind(means,meansCalc1,meansCalc2)
```

```
##              means meansCalc1 meansCalc2
## contVar1  12.27407  12.025310  12.025310
## contVar2  22.44599  22.648520  22.648520
## contVar3  22.18549  22.348987  22.348987
## contVar4  22.46759  22.402189  22.402189
## contVar5  27.21831  27.259461  27.259461
## contVar6  22.80621  22.323295  22.323295
## contVar7  10.18992   9.881249   9.881249
## contVar8  14.65101  14.690074  14.690074
## contVar9  23.32168  23.180793  23.180793
## contVar10 20.28502  20.238427  20.238427
## contVar11 23.87183  23.850068  23.850068
## contVar12 20.89950  20.950427  20.950427
## contVar13 15.65467  15.724085  15.724085
## contVar14 28.46867  28.933707  28.933707
## contVar15 15.84632  15.823120  15.823120
## contVar16 26.74591  26.325279  26.325279
## contVar17 15.72447  15.512789  15.512789
## contVar18 15.33642  15.515943  15.515943
## contVar19 13.73446  14.436562  14.436562
## contVar20 14.64452  14.392330  14.392330
```

```
table(meansCalc1 == meansCalc2)
```

```
##
## TRUE
##   20
```

You can see that changing the margin changes the dimensions of our output:

```
# Calculate the average of each row
rowMeanCalc <- apply(randomData2,MARGIN = 1,FUN = mean)
rowMeanCalc
```

```
##   [1] 18.81440 19.27149 19.34647 20.11105 19.30046 18.78284 20.11789 18.77118
##   [9] 19.38706 19.49253 20.32453 20.52604 18.10251 19.51508 19.06821 19.23547
##  [17] 20.57828 19.68729 19.38488 19.68317 19.42163 19.27084 19.43069 19.57118
##  [25] 18.80541 19.14848 19.96103 18.78698 19.14905 20.15764 18.31221 19.32369
##  [33] 19.90566 19.79821 20.05305 20.19139 19.31772 17.65365 20.22286 19.59169
##  [41] 19.60392 18.93573 18.93560 19.45553 19.45373 19.69267 19.08101 19.20042
##  [49] 19.50227 20.14793 19.42541 19.05196 19.90143 18.80104 19.04737 20.24065
##  [57] 19.43630 19.55147 19.64467 18.35893 19.00613 18.82268 19.47647 20.05867
##  [65] 19.91343 20.13541 18.76509 18.50069 19.28996 19.52210 19.36532 19.98377
##  [73] 19.09369 20.24491 20.30700 20.07853 20.62423 19.92986 19.16553 18.90956
##  [81] 18.54757 19.07192 19.12449 20.20156 19.13124 19.49442 19.68153 19.39881
##  [89] 19.00771 18.78541 19.85630 19.15882 19.14500 19.22776 19.28820 19.15784
##  [97] 19.81432 19.49021 18.96961 19.52540
```

We can also apply functions we have defined ourselves, like the function we defined above to summarize continuous variables.

```
contSummary
```

```
## function (x, normal = T)
## {
##     if (normal == T) {
##         myMean <- round(mean(x), 2)
##         mySD <- round(sd(x), 2)
##         paste0(myMean, " (", mySD, ")")
##     }
##     else {
##         myMedian <- round(median(x))
##         myIQR1 <- round(quantile(x, 1/4), digits = 2)
##         myIQR2 <- round(quantile(x, 3/4), digits = 2)
##         paste0(myMedian, " [", myIQR1, ", ", myIQR2, "]")
##     }
## }
## <bytecode: 0x000001dc324310a8>
```

Notice that the only input required for this function is a single array. Importantly, to use the apply function it can only take input as an array because you will only ever be inputting an array of rows or an array of columns. Now we can use the apply function to calculate summary statistics as follows:

```
apply(randomData2,MARGIN = 2,FUN = contSummary)
```

```
##       contVar1       contVar2       contVar3       contVar4       contVar5
## "12.03 (1.81)" "22.65 (1.97)" "22.35 (1.17)"  "22.4 (0.72)"  "27.26 (1.7)"
##       contVar6       contVar7       contVar8       contVar9      contVar10
## "22.32 (3.73)"  "9.88 (2.74)" "14.69 (4.39)" "23.18 (4.01)" "20.24 (0.74)"
##      contVar11      contVar12      contVar13      contVar14      contVar15
## "23.85 (2.14)" "20.95 (1.76)" "15.72 (1.81)" "28.93 (2.62)" "15.82 (1.21)"
##      contVar16      contVar17      contVar18      contVar19      contVar20
## "26.33 (3.78)" "15.51 (1.42)" "15.52 (1.76)" "14.44 (5.26)" "14.39 (3.99)"
```

You can also define function in line, although for anything complex it is best practice to define it as a function up front.

For example, if I am working with gene expression data, I might want to log2 normalize it prior to calculating my summary statistics.

```
apply(randomData2,MARGIN = 2,FUN = function(x) {log2(mean(x))})
```

```
##  contVar1  contVar2  contVar3  contVar4  contVar5  contVar6  contVar7  contVar8
##  3.588002  4.501345  4.482138  4.485568  4.768685  4.480478  3.304693  3.876770
##  contVar9 contVar10 contVar11 contVar12 contVar13 contVar14 contVar15 contVar16
##  4.534858  4.339025  4.575921  4.388908  3.974904  4.854679  3.983962  4.718377
## contVar17 contVar18 contVar19 contVar20
##  3.955386  3.955679  3.851655  3.847228
```

Or, if you want to define additional inputs for a function you can do it like this:

```
apply(randomData2,MARGIN = 2,FUN = function(x) {contSummary(x,normal = F)})
```

```
##             contVar1             contVar2             contVar3             contVar4
## "12 [10.67, 13.09]" "23 [21.63, 23.75]" "22 [21.55, 23.02]" "22 [21.98, 22.84]"
##             contVar5             contVar6             contVar7             contVar8
## "27 [25.96, 28.31]" "23 [19.82, 24.84]"  "10 [8.17, 11.75]" "14 [12.08, 17.28]"
##             contVar9            contVar10            contVar11            contVar12
## "23 [21.04, 26.16]"  "20 [19.75, 20.7]" "24 [22.39, 25.15]" "21 [19.67, 22.27]"
##            contVar13            contVar14            contVar15            contVar16
## "16 [14.59, 16.86]" "29 [27.47, 30.54]"  "16 [15.06, 16.6]" "26 [23.69, 29.08]"
##            contVar17            contVar18            contVar19            contVar20
##  "16 [14.58, 16.5]"  "15 [14.3, 16.76]" "14 [11.73, 17.71]" "14 [11.38, 16.82]"
```

Correlation Plots

For the remainder of today's class, we are going to use simulated extracellular vesicle (EV) miRNA data which you can download on canvas. This data is currently stored as raw counts from array data in a 798x25 matrix where each row indicates a miRNA probe and each column represents a single sample.

```
# Load data (available on canvas)
load('EVmiRNA.RData')

# Look at top values of data
head(miRNA)
```

```
##                   Subject1   Subject2   Subject3  Subject4  Subject5  Subject6
## hsa-let-7a-5p 0.60028416 45.5374250 39.8202001 0.5439724 1.7036418 0.8218014
## hsa-let-7b-5p 2.28632302  3.5313938  0.1421701 3.7409437 0.7435480 1.6051583
## hsa-let-7c-5p 1.56967081  3.6978740  1.5401812 8.6440336 0.2078016 0.5285168
## hsa-let-7d-5p 0.45013153  0.5316744  0.4572892 1.5572539 2.4809321 3.6656144
## hsa-let-7e-5p 1.65524839  0.9092910  4.9291692 3.0554547 5.4549300 3.8512010
## hsa-let-7f-5p 0.02746762  4.8458211  0.5312497 0.7219376 1.6886609 4.1652172
##                 Subject7  Subject8  Subject9 Subject10 Subject11  Subject12
## hsa-let-7a-5p  1.4268875 1.1644916 0.4477190 3.7706275  2.341564 50.4426852
## hsa-let-7b-5p 45.6244158 3.6671706 0.5912646 0.7170687  3.470848  2.8020564
## hsa-let-7c-5p  2.6131218 3.0809774 0.1867591 2.8857151  5.009104  3.3835651
## hsa-let-7d-5p  2.4877372 3.6752958 1.9517596 1.9679624  2.514447  0.2454894
## hsa-let-7e-5p  3.3131662 6.0011013 1.2977235 5.6647756  1.122553  1.8065104
## hsa-let-7f-5p  0.1229824 0.7641041 0.5531441 2.2793038  2.354166  3.5117765
##               Subject13  Subject14 Subject15  Subject16  Subject17 Subject18
## hsa-let-7a-5p 3.1584249 1.81103856 1.0506802 1.21402785 0.09916022  1.615196
## hsa-let-7b-5p 9.3599591 3.56422244 3.1126187 3.67909172 3.74872164  3.046263
## hsa-let-7c-5p 1.2296840 1.67335225 1.4994374 2.59342468 2.80964038  1.162579
## hsa-let-7d-5p 2.1659421 0.04532317 1.9140693 0.02586989 3.06435526  2.603185
## hsa-let-7e-5p 0.4707412 3.12133496 3.5116455 1.97381795 0.54880816  1.179981
## hsa-let-7f-5p 0.8429741 1.45679080 0.8505713 0.23845618 2.07801264  4.025789
##                Subject19 Subject20 Subject21   Subject22 Subject23  Subject24
## hsa-let-7a-5p 3.70621495  1.103301 6.0568872 0.730431418 0.7666557  0.1576427
## hsa-let-7b-5p 0.04944034  2.477991 1.8124237 0.002790862 1.5003565 28.2966118
## hsa-let-7c-5p 0.41234710  2.419353 2.5114150 4.673653944 3.3274686  2.5063161
## hsa-let-7d-5p 3.90689458  1.695993 0.7975137 4.551423709 0.4894029  3.5985524
## hsa-let-7e-5p 1.78129519  6.956832 5.1397975 1.576929874 5.4476960  3.8055454
## hsa-let-7f-5p 1.27046970  4.433773 0.1432026 0.967950001 3.6744989  0.5287363
##               Subject25
## hsa-let-7a-5p  4.272345
## hsa-let-7b-5p  1.263552
## hsa-let-7c-5p  1.194068
## hsa-let-7d-5p  1.661977
## hsa-let-7e-5p  2.386187
## hsa-let-7f-5p  1.401317
```

```r
# Look at top 20 row (aka miRNA) names
row.names(miRNA)[1:20]
```

```
##  [1] "hsa-let-7a-5p"                 "hsa-let-7b-5p"
##  [3] "hsa-let-7c-5p"                 "hsa-let-7d-5p"
##  [5] "hsa-let-7e-5p"                 "hsa-let-7f-5p"
##  [7] "hsa-let-7g-5p"                 "hsa-let-7i-5p"
##  [9] "hsa-miR-1-3p"                  "hsa-miR-1-5p"
## [11] "hsa-miR-100-5p"                "hsa-miR-101-3p"
## [13] "hsa-miR-103a-3p"               "hsa-miR-105-5p"
## [15] "hsa-miR-106a-5p+hsa-miR-17-5p" "hsa-miR-106b-5p"
## [17] "hsa-miR-107"                   "hsa-miR-10a-5p"
## [19] "hsa-miR-10b-5p"                "hsa-miR-1178-3p"
```

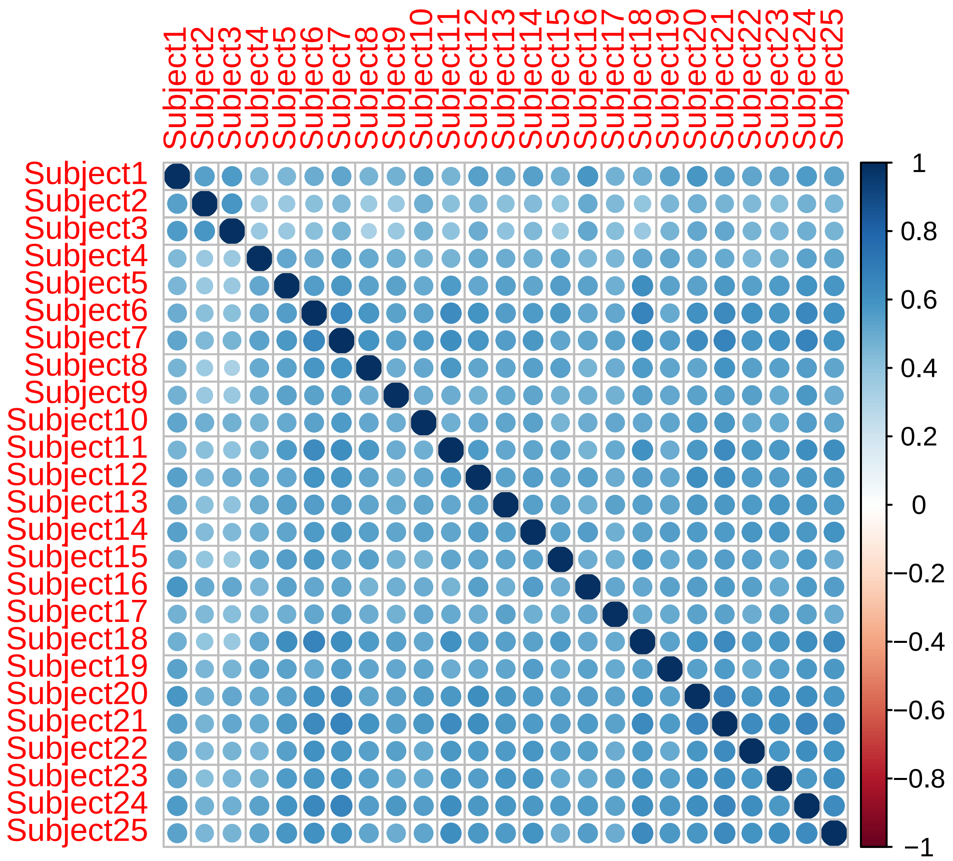
We can calculate correlations for any case in which we have paired data. For example, we can look at the correlation between any two miRNAs in our data set between subjects. We can calculate correlations for a data frame using the *cor()* function in R. When you use the *cor()* function, you will need to designate the method you want it to use. Typically, you will see Pearson Correlation Coefficient used for normally distributed data as it assumes a linear relationship between your two variables and you will see a Spearman's Rank Correlation Coefficient used for non-normal data as it is a non-parametric test that does not assume linearity.

The *cor()* function takes the input of a data frame and will calculate the correlation between each column of your data frame. So, if we use our data frame where columns reflect samples, it will tell us the correlation between each sample.

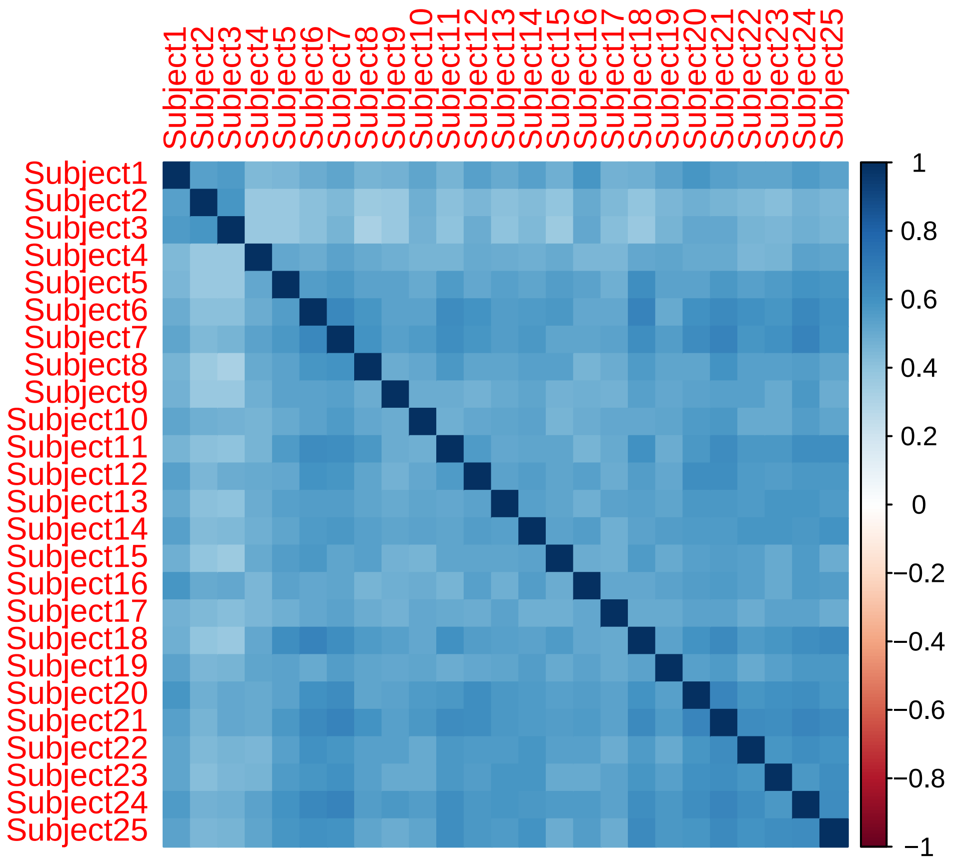
```r
#install.packages('corrplot')
library(corrplot)
```

```
## corrplot 0.95 loaded
```

```r
# Calculate the spearman correlation coefficient
corrMatrix <- cor(miRNA,method = 'spearman')
# Generate a circle-based plot
corrplot(corrMatrix,method = 'circle')
```

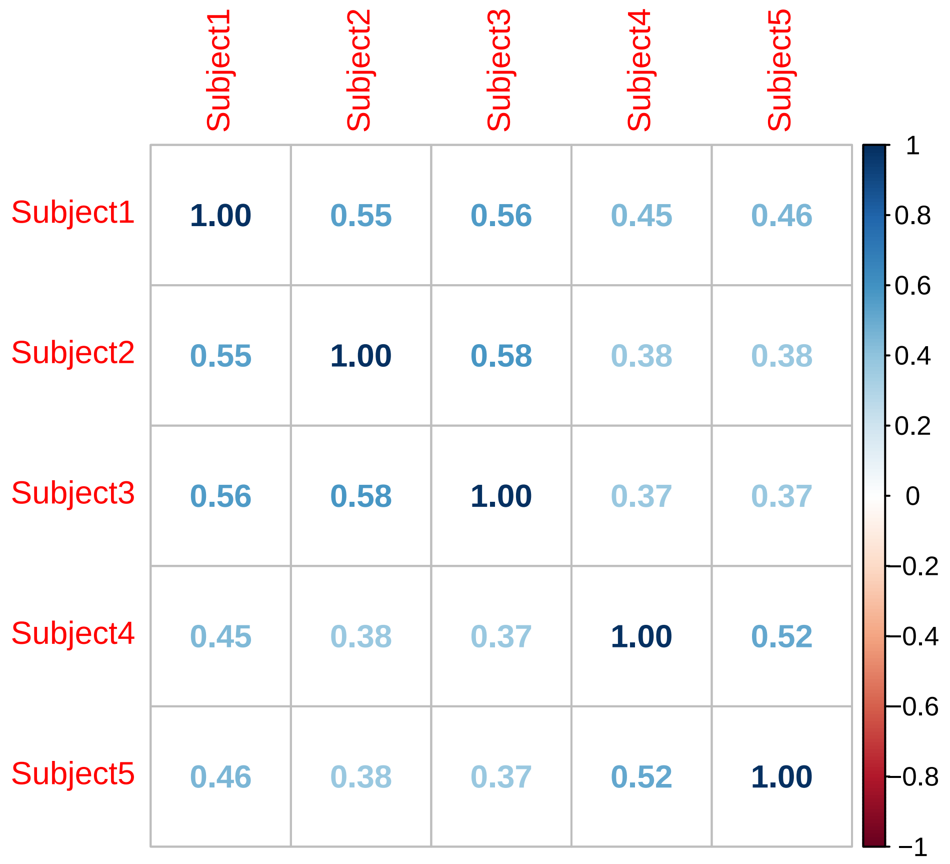


```r
# Generate a color-based correlation plot
corrplot(corrMatrix,method = 'color')
```



We can also produce a plot with the individual correlations included, but it's easier to see this with a smaller plot.

```
# Generate plot with absolute correlations
corrplot(corrMatrix[1:5,1:5],method = 'number')
```



	Subject1	Subject2	Subject3	Subject4	Subject5
Subject1	1.00	0.55	0.56	0.45	0.46
Subject2	0.55	1.00	0.58	0.38	0.38
Subject3	0.56	0.58	1.00	0.37	0.37
Subject4	0.45	0.38	0.37	1.00	0.52
Subject5	0.46	0.38	0.37	0.52	1.00

If we want to generate the correlations between each miRNA we first need to transpose the matrix (i.e. make the columns the rows and the rows the columns).

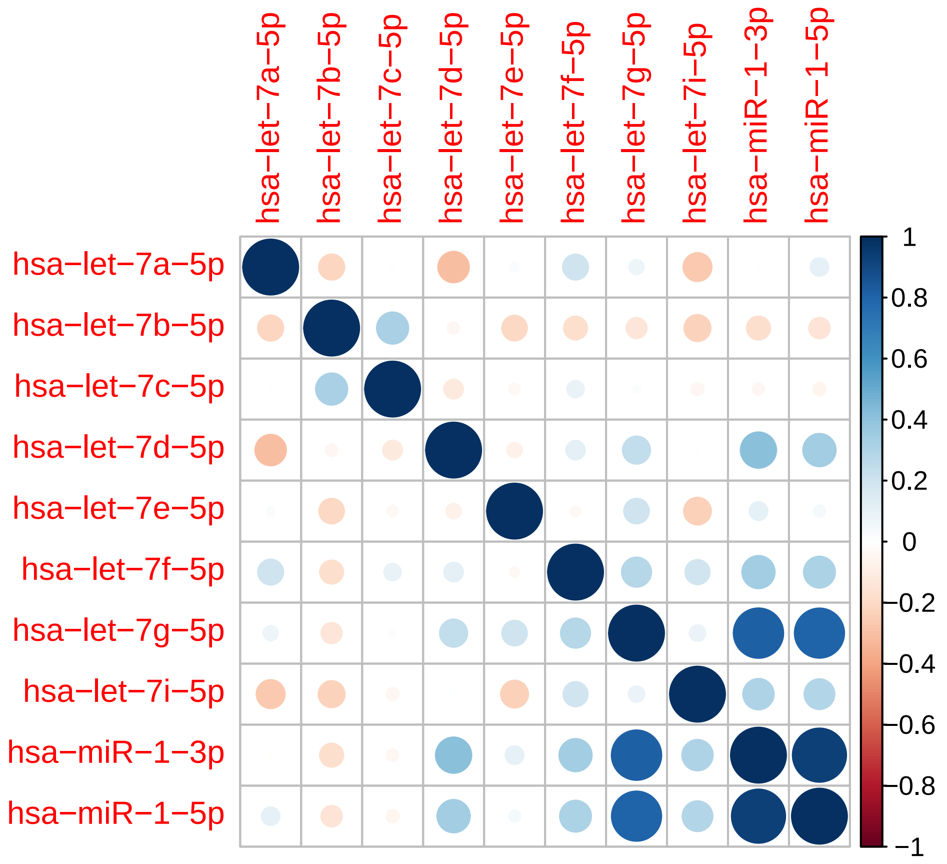
```
# Transpose a subset of the matrix
t(miRNA[1:5,1:5])
```

```
##            hsa-let-7a-5p hsa-let-7b-5p hsa-let-7c-5p hsa-let-7d-5p hsa-let-7e-5p
## Subject1       0.6002842     2.2863230     1.5696708     0.4501315      1.655248
## Subject2      45.5374250     3.5313938     3.6978740     0.5316744      0.909291
## Subject3      39.8202001     0.1421701     1.5401812     0.4572892      4.929169
## Subject4       0.5439724     3.7409437     8.6440336     1.5572539      3.055455
## Subject5       1.7036418     0.7435480     0.2078016     2.4809321      5.454930
```

Here, we can see our miRNA are now in the columns, so that when we use the *cor()* function it will calculate between-miRNA correlations instead of between sample correlations.

```
# Calculate correlations
miRNA.cor <- cor(t(miRNA),method = 'spearman')

# Plot correlations
corrplot(miRNA.cor[1:10,1:10],method = 'circle')
```



Heat Maps

Heat maps are an easy way to visualize data and look for overall trends when working with highly dimensional data. There are many packages you can use to generate heatmaps but today we're going to work with *pheatmap* which has really easy functionality.

To start, we're going to identify miRNA in our data that have the most variability so that we can have a more interesting heatmap to look at. Variance is just a measure of how much values within a set of numbers differ from the mean. So, the set {3,3.5,4} would have a much lower variance than the set {0,2.5,8} even though they both have the same mean. The formula for variance looks like this:

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

In R, we can calculate this using the *var()* function:

```r
# Compare sample variances
var(c(3,3.5,4))
```

```
## [1] 0.25
```

```r
var(c(0,2.5,8))
```

```
## [1] 16.75
```

Notably, standard deviation is just the square root of variance, which we can see easily here.

```
# Calculate variance
var(c(0,2.5,8))
```

```
## [1] 16.75
```

```
# Calculate the sd squared
sd(c(0,2.5,8))^2
```

```
## [1] 16.75
```

Now, we can use the apply function to calculate these values quickly for all the 798 miRNA in our dataset.

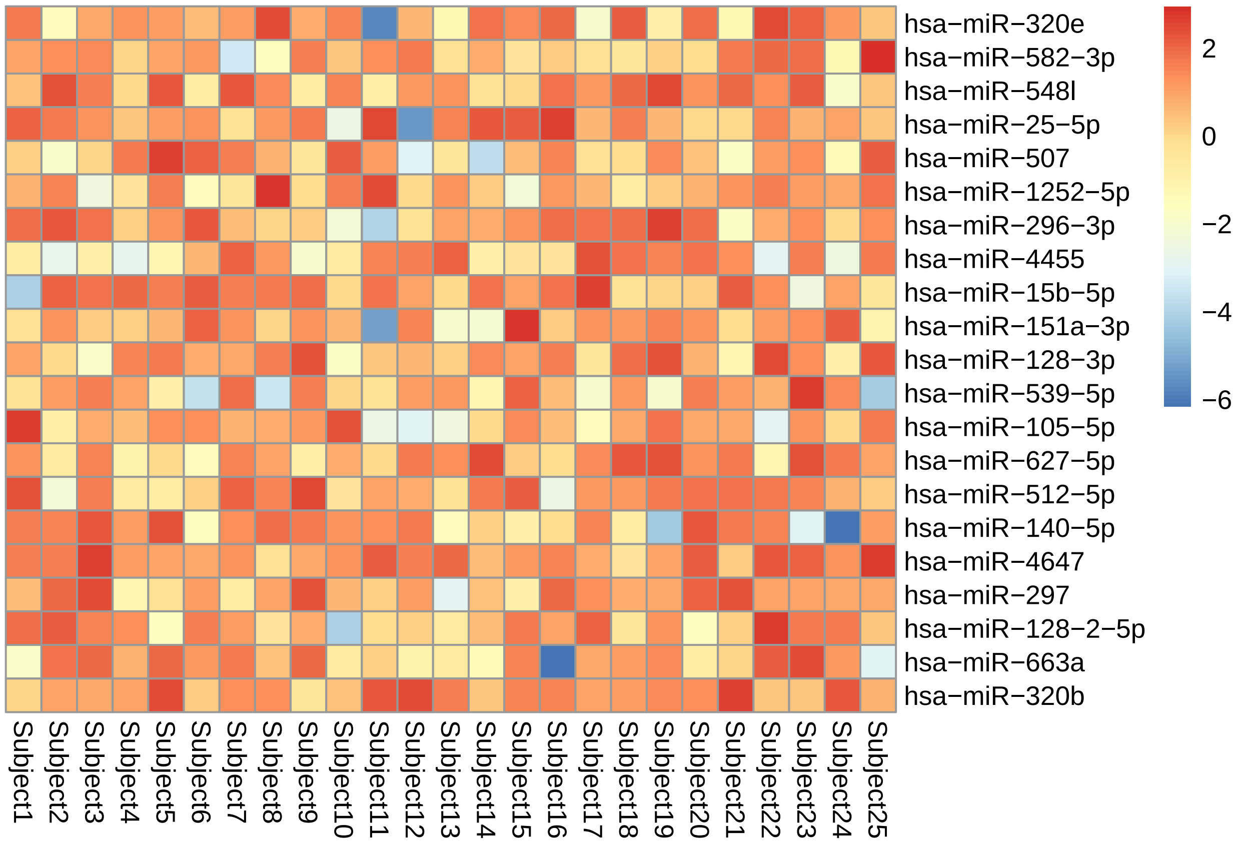
```
# Calculate variance of each miRNA
variance <- apply(miRNA,MARGIN = 1,FUN = var)
# Order rows of miRNA so that highest variance in expression is on top
miRNA2 <- miRNA[order(variance,decreasing = T),]

# Log2-normalize data for plotting
log2.miRNA <- log2(miRNA2)
```

Now that we have our data ready, we can generate a very simple heatmap.

```
#install.packages('pheatmap')
library(pheatmap)

# Generate heatmap without clustering
pheatmap(log2.miRNA[700:720,],
         cluster_rows = F,
         cluster_cols = F)
```



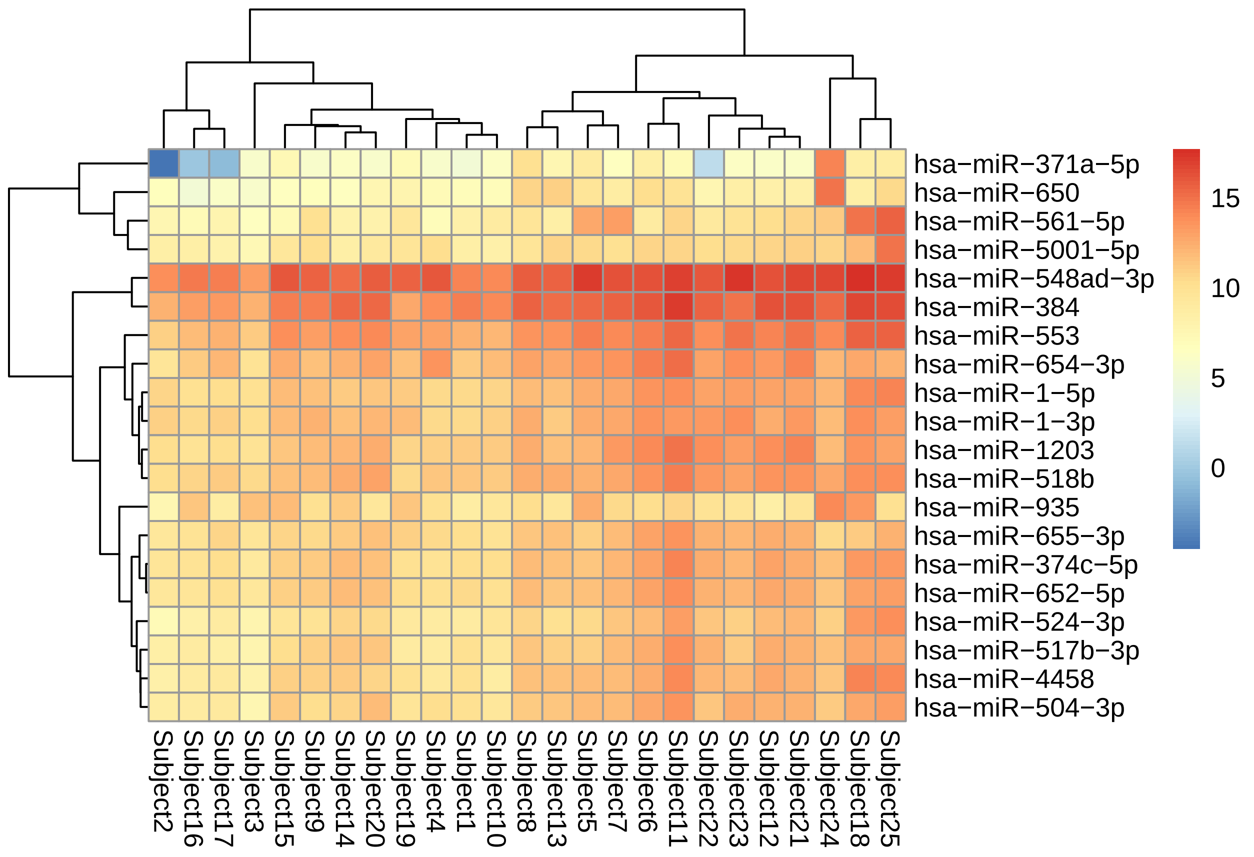
Basic Hierarchical Clustering

Clustering is a way of identifying how similar different samples, etc. are from each other in our data. We often add clustering to heat maps to help us visualize how similar vs. dissimilar different rows and columns are from one another. There are lots of different algorithms you can use for clustering but for the sake of this lecture, we're just going to talk about Euclidean distance as that is a common default for heatmaps. The equation for Euclidean distance looks like this:

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

We can calculate this value between every combination of samples or miRNAs in our data and then create a dendrogram by designating which values are closer to each other, or less close to each other. Every time you assign one set of samples as closer to each other, you will calculate distance again between that now cluster of samples, and all other remaining samples, iteratively until all samples have been linked. Ultimately, the end product looks like this:

```r
# Add clusters
# Note: euclidean is the default so as long as clustering is turned on this is what you will get)
pheatmap(log2.miRNA[1:20,],
         clustering_distance_cols = 'euclidean',
         clustering_distance_rows = 'euclidean')
```

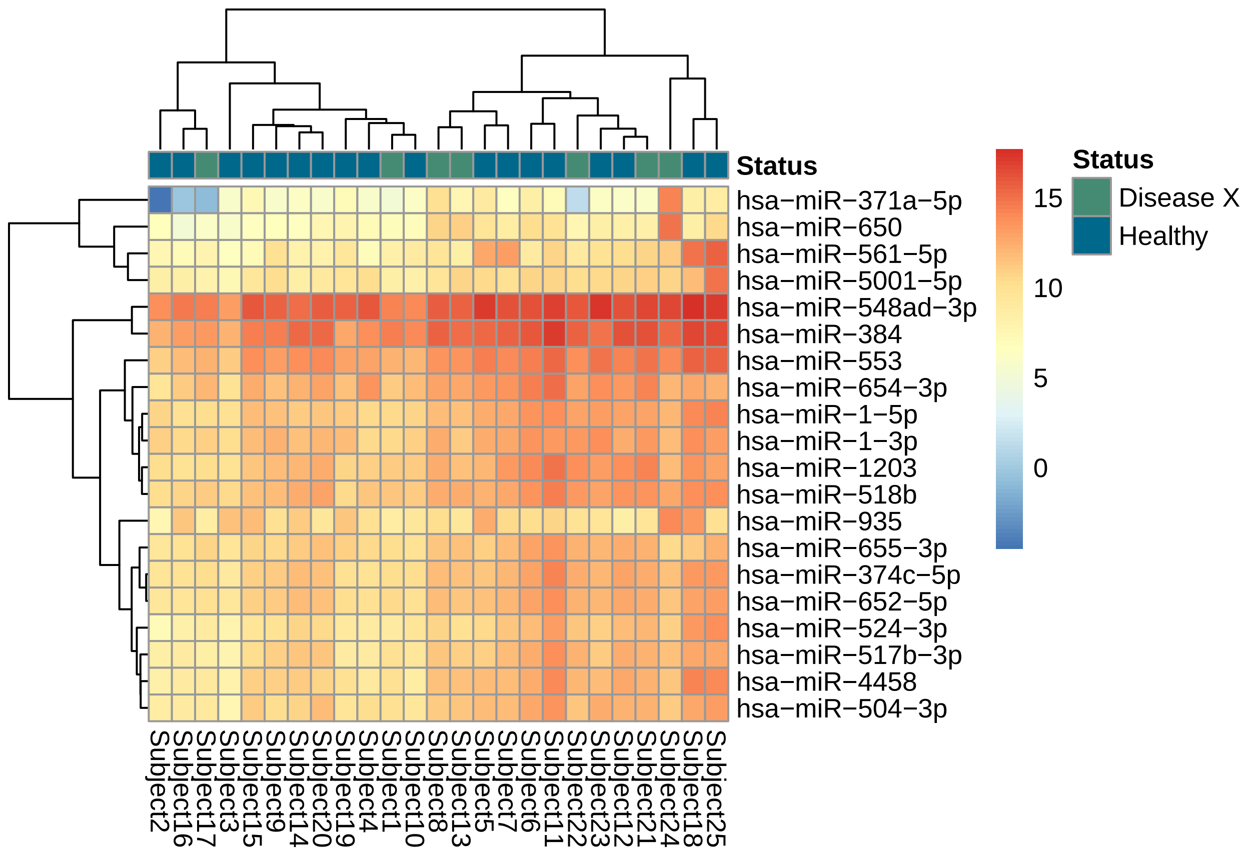


An important thing to remember when ever you're looking at a clustering dendrogram is that not all order matters. Technically, any point it branches off it can pivot on that access so ultimately, Subject10 could end up next to Subject8, Subject21, Subject25, or any of the other members of that cluster.

Now we can start to see our samples breaking out into 2 primary clusters. We may want to visualize additional things, such as if this clustering pattern is related to a covariate of interest.

```r
# Define covariate for tracking bar
set.seed(9876)
annotationData <- data.frame(row.names = colnames(miRNA),
                             'Status' = c(factor(rbinom(n = 25,size = 1,prob = 0.6),
                                                 labels = c('Disease X','Healthy'))))
annotationColors <- list(Status = c('Disease X' = 'aquamarine4',
                                    'Healthy' = 'deepskyblue4'))

# Generate heatmap
pheatmap(log2.miRNA[1:20,],
         clustering_distance_cols = 'euclidean',
         clustering_distance_rows = 'euclidean',
         annotation_col = annotationData,
         annotation_colors = annotationColors)
```



In Class Acitivity

1. Write a function to calculate the relative abundance of each miRNA in each sample (i.e. what percentage of the total miRNA in a given sample is your miRNA of interest). Verify that each sample has a total relative abundance of 1.

2. Using the apply function, identify the highest relative abundance each miRNA has in a single sample.

3. Sort the dataset by miRNA with the highest relative abundance and generate a heatmap of the relative abundance (not the absolute counts) of each miRNA, including the top 20 miRNA by single-sample relative abundance.

4. Generate a random binary variable for sex and a categorical variable for age group using distributions and age cutoffs (hint: use the *cut()* function) of your chosing. Add tracking bars to your plot for your generated variables.