Week 11 Module 3 – Sub-Group Analysis

CSCI E-5a: Programming in R

Let's clear the global computing environment:

```
rm( list = ls() )
```

Module Preview

Hello! And welcome to Module 3: Sub-Group Analysis.

In this module we will investigate another powerful method for summarizing data, called a sub-group analysis.

- In Section 1, we'll define the concept of a sub-group analysis.
- In Section 2, we'll perform a sub-group analysis by hand.
- In Section 3, we'll learn how to automate the sub-group analysis.
- In Section 4, we'll use the `split()` function to split the values in a vector based on the levels of a factor.
- In Section 5, we'll learn how to use the `tapply()` function to perform sub-group anlayses.
- In Section 6, we'll learn how to use the `aggregate()` function to perform sub-group analyses.

When you've completed this module, you'll be able to:

- Explain the concept of a sub-group analysis
- Perform a sub-group analysis by hand
- Automate a sub-group analysis
- Use the `split()` function to split the values in a vector based on the levels of a factor
- Use the `tapply()` function to perform sub-group anlayses.
- Use the `aggregate()` function to perform sub-group analyses.

There are 3 new built-in R functions in this module:

- `split()`
- `tapply()`
- `aggregate()`

All right! Let's start out by defining the concept of a sub-group analysis.

Section 1: Definition

> **Main Idea:** *We can define the concept of a sub-group analysis*

In this section, we'll define the concept of a sub-group analysis.

In this module, we're going to learn how to conduct what's called a "sub-group" analysis, and how to visualize the results.

Sometimes this can be a little confusing for people, so I want to go through it step-by-step.

The best way to learn this concept is to study a concrete example, so let's return to the `iris` data frame, which is built-in to R.

```
head( iris )
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Recall that this data frame consists of 150 measurements on 50 plants each for the *setosa*, *versicolor*, and *virginica* species.

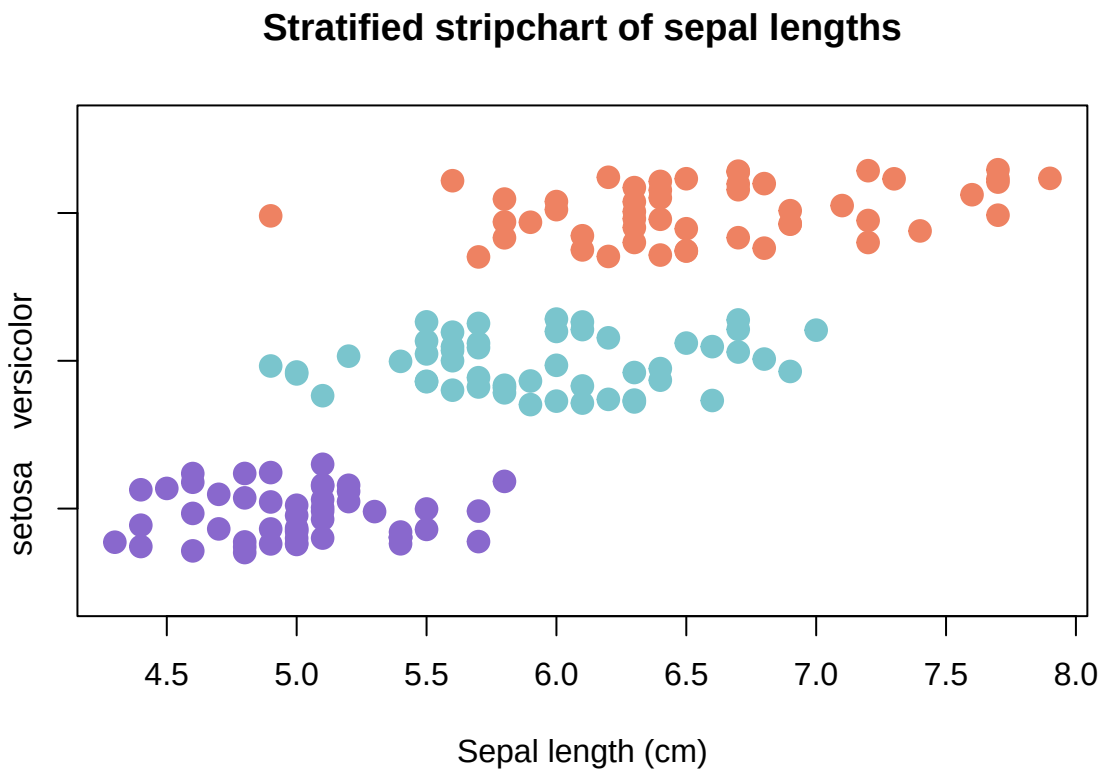
These three species form a factor variable, named `species`.

Then there are four numerical measurements, one of which is the length of the sepal.

This variable is named `Sepal.Length`.

Let's make a stratified stripchart of the `Sepal.Length` values in the `iris` data frame, across the three levels of the `species` factor:

```
stripchart(
    iris$Sepal.Length ~ iris$Species,
    main = "Stratified stripchart of sepal lengths",
    xlab = "Sepal length (cm)",
    method = "jitter",
    jitter = 0.3,
    pch = 19,
    cex = 1.5,
    col =
        c( "mediumpurple3", "cadetblue3", "salmon2")
)
```



When we examine this stratified stripchart, we can draw some simple conclusions:

- First, the values of `Sepal.Length` for the *setosa* observations are on average less than the values of `Sepal.Length` for the *versicolor* and *virginica* species.
- Second, the values of `Sepal.Length` for the *versicolor* observations are on average less than the values of `Sepal.Length` for the *virginica* species.

Of course, it's important to remember the phrase "on average" – there are `Sepal.Length` values for some *setosa* plants that are larger than the `Sepal.Length` values for some *versicolor* and *virginica* plants.

But on average the `Sepal.Length` values for the *setosa* plants are smaller than the `Sepal.Length` values for the *versicolor* plants, and these are on average smaller than the `Sepal.Length` values for the *virginica* plants.

How can we summarize these ideas numerically?

One way to do this is to create some sort of tabular display where each row displays a species and its sample mean for the `Sepal.Length` values.

This is called a "sub-group" analysis, because we're calculating the sample mean of the `Sepal.Length` values for each species (i.e. sub-group).

This module is focused on how to calculate such sub-group analyses.

So that's how to define the concept of a sub-group analysis.

Now let's see how to perform a sub-group analysis by hand.

Section 2: Sub-Group Analysis

> **Main Idea:** *We can perform a sub-group analysis by hand*

In this section, we'll perform a sub-group analysis by hand.

First, let's be absolutely clear about exactly what this "sub-group" analysis is.

We'll start by implementing this by hand.

This is not the best way to work in practice, and we're doing it just as a way for you to understand the procedure.

This way, when you do see the elegant tool for doing this, you'll know exactly what's going on, and appreciate all the labor that it saves you.

That is, we want to calculate the sample mean of the `Sepal.Length` values for the *setosa* observations, the sample mean of the `Sepal.Length` values for the *versicolor* values, and the sample mean of the `Sepal.Length` values for the *virginica* values.

There's a straightforward strategy for doing this:

- First, we select the values of `Sepal.Length` that correspond to the *setosa* species, calculate the sample mean of this filtered vector, and then store the result in a variable.
- Next, we select the values of `Sepal.Length` that correspond to the *versicolor* species, calculate the sample mean of this filtered vector, and then store the result in a variable.
- Finally, we select the values of `Sepal.Length` that correspond to the *virginica* species, calculate the sample mean of this filtered vector, and then store the result in a variable.

We can then display these three values.

Let's get to work!

First, we'll select all the *setosa* values:

```
setosa.sepal.length.data <-
    iris$Sepal.Length[ iris$Species == "setosa" ]

setosa.sepal.length.sample.mean <-
    mean( setosa.sepal.length.data )

cat(
    "Setosa sepal length sample mean:",
    formatC(
        setosa.sepal.length.sample.mean,
        format = "f",
        digits = 2
    )
)
```

```
## Setosa sepal length sample mean: 5.01
```

First, we'll select all the *versicolor* values:

```
versicolor.sepal.length.data <-
    iris$Sepal.Length[ iris$Species == "versicolor" ]

versicolor.sepal.length.sample.mean <-
    mean( versicolor.sepal.length.data )

cat(
    "Versicolor sepal length sample mean:",
    formatC(
        versicolor.sepal.length.sample.mean,
        format = "f",
        digits = 2
    )
)
```

```
## Versicolor sepal length sample mean: 5.94
```

Finally, we'll select all the *virginica* values:

```
virginica.sepal.length.data <-
    iris$Sepal.Length[ iris$Species == "virginica" ]

virginica.sepal.length.sample.mean <-
    mean( virginica.sepal.length.data )

cat(
    "Virginica sepal length sample mean:",
    formatC(
        virginica.sepal.length.sample.mean,
        format = "f",
        digits = 2
    )
)
```

```
## Virginica sepal length sample mean: 6.59
```

Now let's put all of this into a named vector:

```
sepal.length.sub.group.analysis.vector <-
    c(
        "setosa" =
            setosa.sepal.length.sample.mean,
        "versicolor" =
            versicolor.sepal.length.sample.mean,
        "virginica" =
            virginica.sepal.length.sample.mean
    )
```

Now we can display this named vector directly:

```
sepal.length.sub.group.analysis.vector
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

If we want to, we can convert this named vector to a table:

```
as.table( sepal.length.sub.group.analysis.vector )
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

We can also put these values into a data frame:

```
subgroup.analysis.data.frame <-
  data.frame(
    Species =
      c( "Setosa", "Versicolor", "Virginica" ),
    Sample.Mean =
      round(
        c( setosa.sepal.length.sample.mean,
           versicolor.sepal.length.sample.mean,
           virginica.sepal.length.sample.mean ),
        digits = 1
      )
  )

names( subgroup.analysis.data.frame )[ 2 ] <- "Sample Mean"

subgroup.analysis.data.frame
```

```
##      Species Sample Mean
## 1     Setosa         5.0
## 2 Versicolor         5.9
## 3  Virginica         6.6
```

So that's how to peform a sub-group analysis by hand.

Now let's learn how to automate this sub-group analysis.

Section 3: Automating the Sub-Group Analysis

> **Main Idea:** *We can automate this sub-group analysis*

In this section, we'll learn how to automate the sub-group analysis.

You might have noticed that the method we used for conducting the sub-group analysis was very repetitive.

For each species, we had to select the values from the data frame, calculate the sample mean, and then store this result.

This lead to a highly repetitive process, with a lot of duplicated code.

We can automate this by writing a `for()` loop.

First, we construct a factor consisting of unique representatives of each species:

```
species.names.vector <-
  unique( iris$Species )

species.names.vector
```

```
## [1] setosa     versicolor virginica
## Levels: setosa versicolor virginica
```

Next, we construct a storage vector of the same length as the number of different species, and we assign the unique representatives of the species as the names of the vector.

```
subgroup.analysis.vector <-
  numeric( length( species.names.vector ) )

names( subgroup.analysis.vector ) <-
  species.names.vector

subgroup.analysis.vector
```

```
##     setosa versicolor  virginica
##          0          0          0
```

Now we can iterate over the names of the species:

- For each species, we first select the rows of the `iris` data frame that correspond to that particular species.
- Then we calculate the sample mean of the `Sepal.Length` values.
- Finally, we store this sample in the storage vector using the name.

When we're all done with this process, we'll have a vector populated with the sample mean for each species.

```
for( current.species in species.names.vector ) {

  current.species.data.frame <-
    iris[ iris$Species == current.species, ]

  current.species.sepal.length.sample.mean <-
    mean( current.species.data.frame$Sepal.Length )

  subgroup.analysis.vector[ current.species ] <-
    current.species.sepal.length.sample.mean
}

subgroup.analysis.vector
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

So that's how to automate the sub-group analysis.

Now let's see how to use the `split()` function to partition the values in a vector based on the levels of a factor.

Section 4: The `split()` Function

> **Main Idea:** *We can partition the values in a vector based on the levels of a factor*

In this section, we'll use the `split()` function to partition the values in a vector based on the levels of a factor.

Most of the work involved in doing a subgroup analysis involves partitioning the values in one vector based on the categorical data in another vector or factor.

There's actually a built-in R function for this process called the `split()` function.

The `split()` function takes two required input arguments:

- First, a vector of numeric values, denoted `x`.
- Second, a vector or factor of categorical data, denoted `f`.

The `split()` function then returns the subvectors of `x`, partitioned on the basis of the categorical data in `f`.

Thus, for our example, we partitioned the numeric vector of `Sepal.Length` values based on the categorical data in the `Species` factor.

The `split()` function will take the `Sepal.Length` vector and then will return the subvectors of this numeric vector, with one vector for each different species.

Before we move on, can you guess what class of data structure MUST be returned by `split()`?

Think about this carefully – in our example, we want to split the `Sepal.Length` data into 3 distinct subvectors, so we have to use a class that can bundle multiple vectors together.

The only R data structure of sufficient generality to able to bundle vectors together is the list structure.

So `split()` must return a list.

Let's try this out:

```
split(
  x = iris$Sepal.Length,
  f = iris$Species
)
```

```
## $setosa
##  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1 5.7
## [20] 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0 5.5 4.9
## [39] 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0
##
## $versicolor
##  [1] 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2
```

```
## [20] 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3
## [39] 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7
##
## $virginica
##  [1] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [20] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4
## [39] 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

So the `split()` function has done a lot of work for us!

Now we can just select each element from this named list, calculate the sample mean of the values in the subvector, and store this result in the storage vector.

So that's how to partition the values in a vector based on the levels of a factor.

Now let's see how to use the `tapply()` function to perform sub-group analyses.

Section 5: The `tapply()` Function

> **Main Idea:** *We can use the `tapply()` function to perform a sub-group analysis*

In this section, we'll learn how to use the `tapply()` function to perform sub-group analyses.

The `split()` function is very appealing, and it can do a lot of work for us.

However, we still have to calculate the sample mean and store the result in a storage vector.

This analysis is so common that R provides another built-in mechanism for automating the entire process.

The function `tapply()` takes three arguments:

- First, it takes a numeric vector, denoted by `X`.
- Next, it takes a factor, or something that can be coerced to a factor.
- Finally, it takes a function to apply to the numeric vector.

The function `tapply()` automatically performs the filtering operation on the values in the numeric vector for each factor level, and then applies the function to each of these filtered vectors.

Let's try this with our iris data:

```
tapply(
  X = iris$Sepal.Length,
  INDEX = iris$Species,
  FUN = mean
)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

There was something very extraordinary about this example.

We passed the function `mean` to the `tapply()` as an input argument.

We've seen many examples of different kinds of objects being used as function arguments, but this is the first time that we've actually passed a *function* as an argument.

This is quite general, and indeed we can write functions that take other functions as arguments.

Treating functions as regular objects that can be passed to a function is called "functional programming".

This is an advanced technique, and we won't be able to explore it in this course, but you can see that it has great potential to allow you to write very flexible functions.

```
baseball.data.frame <- read.csv( "Baseball Batting Database.csv" )

split(
  baseball.data.frame$H,
  baseball.data.frame$Name
)
```

```
## $`Babe Ruth`
##  [1]   2  29  37  40  95 139 172 204 128 205 200 104 184 192 173 172 186 199 156
## [20] 138 105  13
##
## $`Hank Aaron`
##  [1] 131 189 200 198 196 223 172 197 191 201 187 181 168 184 174 164 154 162 119
## [20] 118  91 109  62
##
## $`Mickey Mantle`
##  [1]  91 171 136 163 158 188 173 158 154 145 163 121  54 141  92  96 108 103
##
## $`Reggie Jackson`
##  [1]  21 138 151 101 157 132 158 146 150 138 150 140 138 154  79 146  77 117 116
## [20] 101  74
##
## $`Roberto Clemente`
##  [1] 121 169 114 150 128 179 201 168 192 211 194 202 209 146 175 145 178 118
##
## $`Ted Williams`
##  [1] 185 193 185 186 176 181 188 194 106 169   4  37 133 114 138 163 135  69  98
##
## $`Willie Mays`
##  [1] 127  30 195 185 171 195 208 180 190 176 189 187 171 177 159 128 144 114 139
## [20] 113   9  52  44
```

So that's how to use the `tapply()` function to perform sub-group analyses.

Now let's learn how to use the `aggregate()` function to perform sub-group analyses.

Section 6: The `aggregate()` function

> **Main Idea:** *We can use the `aggregate()` function to perform a sub-group analysis*

In this section, we'll learn how to use the `aggregate()` function to perform sub-group analyses.

The `tapply()` performs a subgroup analysis or summary, and returns the results in a table data structure.

Instead of a table, it's often more useful to obtain the results in a data frame.

To do this, we can use the `aggregate()` function.

This function works similarly to the `tapply()` function, although you have to pass the input arguments to `aggregate()` using lists.

The `aggregate()` function takes three required input arguments:

- The first argument, denoted `x`, is a *list* of numeric vectors, all of which have to be the same length.
- The second argument, denoted `by`, is a *list* of vectors or factors that we use for partitioning the `x` vector.
- Finally, the third argument, denoted by `FUN`, is a single function.

The `aggregate()` function then works just like `tapply()`: it takes the numeric vectors in `x`, partitions them using the categorical data in `by`, and then applies the function `FUN` to each of the subvectors.

However, `aggregate()` returns the result as a data frame, not as a table structure, which is what `tapply()` does.

Let's try this out with our example:

```
aggregate(
  list( iris$Sepal.Length ),
  list( iris$Species ),
  mean
)
```

```
##      Group.1 c.5.1..4.9..4.7..4.6..5..5.4..4.6..5..4.4..4.9..5.4..4.8..4.8..
## 1     setosa                                                             5.006
## 2 versicolor                                                             5.936
## 3  virginica                                                             6.588
```

Notice that this is now a data frame, not a table.

The numbers that are calculated are correct, but the column headers are awful.

We could fix that by going in and destructively modifying the data frame names.

We could also specify the column names when we run `aggregate()`:

```
aggregate(
  list( Sample.Mean = iris$Sepal.Length ),
  list( Species = iris$Species ),
  mean
)
```

```
##      Species Sample.Mean
## 1     setosa       5.006
## 2 versicolor       5.936
## 3  virginica       6.588
```

```
aggregate.batting.data <-
  aggregate(
  list( "At Bats" = baseball.data.frame$AB, H = baseball.data.frame$H) ,
  list( baseball.data.frame$Name ),
  sum
)
```

```
names( aggregate.batting.data )[ 2 ] <- "At Bats"

aggregate.batting.data
```

```
##            Group.1 At Bats    H
## 1        Babe Ruth    8399 2873
## 2       Hank Aaron   12364 3771
## 3    Mickey Mantle    8102 2415
## 4   Reggie Jackson    9864 2584
## 5 Roberto Clemente    9454 3000
## 6     Ted Williams    7706 2654
## 7      Willie Mays   10881 3283
```

So that's how to use the `aggregate()` function to perform a sub-group analysis.

Now let's review what we've learned in this module.

Module Review

In this module we investigated another powerful method for summarizing data, called a sub-group analysis.

- In Section 1, we defined the concept of a sub-group analysis.
- In Section 2, we performed a sub-group analysis by hand.
- In Section 3, we learned how to automate the sub-group analysis.
- In Section 4, we used the `split()` function to split the values in a vector based on the levels of a factor.
- In Section 5, we learned how to use the `tapply()` function to perform sub-group anlayses.
- In Section 6, we learned how to use the `aggregate()` function to perform sub-group analyses.

Now that you've completed this module, you should be able to:

- Explain the concept of a sub-group analysis
- Perform a sub-group analysis by hand
- Automate a sub-group analysis
- Use the `split()` function to split the values in a vector based on the levels of a factor
- Use the `tapply()` function to perform sub-group anlayses.
- Use the `aggregate()` function to perform sub-group analyses.

There were 3 new built-in R functions in this module:

- `split()`
- `tapply()`
- `aggregate()`

All right! That's it for Module 3: Sub-Group Analysis.

Now let's move on to Module 4: Matrices.