Module 4: supplement

BIOSTAT 213: Introduction to Computing in the R Software Environment

Sunday, February 17, 2019

Subsetting data

The bulk of Module 4 involved **subsetting data**. By subsetting, I mean the process of only keeping some portion (subset) of the data. We discussed two methods for subsetting data: **indexing** (bracket notation) and **the `subset()` function**. We discussed three methods for indexing: indexing using a **logical vector**, indexing using a **numeric vector**, and indexing using a **character vector**.

Subsetting 1-dimensional data

In the case of 1-dimensional data, subsetting means keeping just some of the elements. For example, from slide 9 of Module 4:

```
> animals<-c('giraffe','zebra','elephant','lion')
> animals
[1] "giraffe"  "zebra"    "elephant" "lion"
> animals[animals=='elephant']
[1] "elephant"
```

Here we are indexing **using a logical vector**. Specifically, we are indexing using the logical vector that indicates whether each element of `animals` is equal to `'elephant'`. Let's take a look at that logical vector:

```
> animals=='elephant'
[1] FALSE FALSE  TRUE FALSE
```

Placing this logical vector inside the brackets results in subsetting: we only get the elements of `animals` that are in the `TRUE` position.

Depending only on our goals and imagination, we can put *almost any* logical vector inside the brackets, so long as the logical vector's length matches the length of the vector we are indexing (this length requirement is not strictly true, but recommended for clarity). For example:

```
> animals[nchar(animals)==4]
[1] "lion"
> nchar(animals)==4
[1] FALSE FALSE FALSE  TRUE
```

Alternatively, we can index **using a numeric vector**. For example:

```
> animals
[1] "giraffe"  "zebra"    "elephant" "lion"
> animals[3]
[1] "elephant"
```

Placing the numeric vector `3` inside the brackets results in subsetting: we only get the elements of `animals` that are in the numeric position `3`.

Depending only on our goals and imagination, we can put *almost any* numeric vector inside the brackets. For example, from slide 47 of Module 4:

```
> animals
[1] "giraffe"  "zebra"    "elephant" "lion"
> animals[length(animals)]
[1] "lion"
> length(animals)
[1] 4
```

Finally, we can index using a **character vector**. This will only work on 1-dimensional data objects if the data object has names. For example, from slide 70 of Module 4:

```
> tt<-t.test(ii$bmxbmi~ii$gender)
> tt$estimate
mean in group Female   mean in group Male
            29.65621             28.49810
> class(tt$estimate)
[1] "numeric"
> names(tt$estimate)
[1] "mean in group Female" "mean in group Male"
> tt$estimate['mean in group Male']
mean in group Male
           28.4981
```

Placing the character vector `'mean in group Male'` inside the brackets results in subsetting: we only get the elements of `tt$estimate` named `mean in group Male`.

Unlike with indexing by logical or numeric vectors, we are much more limited in what sorts of character vectors we can put inside brackets: the character vector must include valid names. For example, the following results in `NA` because `mean among the males` is not one of the names of `tt$estimate`:

```
> tt$estimate['mean among the males']
<NA>
  NA
```

Note additionally that 1-dimensional data objects usually do not have names. In such situations, indexing by names is inapplicable.

Do not confuse the values of `animals`, for example, for names:

```
> animals
[1] "giraffe"  "zebra"    "elephant" "lion"
> names(animals)
NULL
> animals['elephant']
[1] NA
```

Here, indexing results in `NA` because `animals` does not have names.

Subsetting 2-dimensional data

In the case of 2-dimensional data, subsetting means keeping just some of the rows (indexing of rows) or just some of the columns (indexing of columns), or both.

Indexing of rows

To keep some rows, the index must be placed *before the comma* because the space before the comma represents rows.

For example, from slide 54 of Module 4:

```
> mm<-lm(bpxdi1~bmxbmi+ridageyr+gender,data=ii)
> ee<-summary(mm)$coefficients
> ee
               Estimate Std. Error    t value     Pr(>|t|)
(Intercept) 62.677125492 0.92890704 67.4740555 0.000000e+00
bmxbmi       0.217201438 0.02586950  8.3960418 5.931366e-17
ridageyr    -0.005565416 0.01022388 -0.5443544 5.862216e-01
genderMale   2.539452656 0.35787328  7.0959549 1.462151e-12
> ee[c(FALSE,TRUE,TRUE,TRUE),]
              Estimate Std. Error    t value     Pr(>|t|)
bmxbmi      0.217201438 0.02586950  8.3960418 5.931366e-17
ridageyr   -0.005565416 0.01022388 -0.5443544 5.862216e-01
genderMale  2.539452656 0.35787328  7.0959549 1.462151e-12
```

Here, are we indexing rows using a **logical vector**. Let's take a look at that logical vector:

```
> c(FALSE,TRUE,TRUE,TRUE)
[1] FALSE  TRUE  TRUE  TRUE
```

Placing the logical vector before the comma results in subsetting of rows: we only get the rows of `ee` that are in the TRUE position.

Depending only on our goals and imagination, we can put *almost any* logical vector before the comma, so long as the logical vector's length matches the number of rows (this length requirement is not strictly true, but recommended for clarity).

Alternatively, we can index rows using a **numeric vector**. For example:

```
> ee[2:3,]
             Estimate Std. Error    t value     Pr(>|t|)
bmxbmi    0.217201438 0.02586950  8.3960418 5.931366e-17
ridageyr -0.005565416 0.01022388 -0.5443544 5.862216e-01
```

Placing the numeric vector `2:3` before the comma results in subsetting: we only get the rows of `ee` that are in the numeric position `2` or `3`.

Depending only on our goals and imagination, we can put *almost any* numeric vector before the comma.

Finally, we can index rows using a **character vector**. For example:

```
> ee[c('ridageyr','bmxbmi'),]
             Estimate Std. Error    t value     Pr(>|t|)
ridageyr -0.005565416 0.01022388 -0.5443544 5.862216e-01
bmxbmi    0.217201438 0.02586950  8.3960418 5.931366e-17
```

Placing the character vector `c('ridageyr','bmxbmi')` before the comma results in subsetting of rows: we only get the rows of `ee` named `ridageyr` or `bmxbmi`.

The `subset()` function

An alternate way to subset rows is to use the `subset()` function. For example, from slide 84 of Module 4:

```
> ss<-subset(ii,ridageyr<=30)
> nrow(ss)
[1] 1073
```

The first argument to `subset()` is the data object you want to subset, while the second argument can be a **logical vector**. The length of the logical vector must match the number of rows. This is entirely *analogous to indexing of rows using a logical vector*. For example, from slide 84 of Module 4:

```
> ss<-subset(ii,ridageyr<=30)
> nrow(ss)
[1] 1073
> ss<-ii[ii$ridageyr<=30,]
> nrow(ss)
[1] 1073
```

A distinction is that with `subset()`, we do not need to use commas to separate dimensions. An additional distinction is that with `subset()` we do not need to worry about missing values. For example, from slide 86 of Module 4:

```
> ss<-subset(ii,bmxwt<=40)
> nrow(ss)
[1] 10
> ss<-ii[!is.na(ii$bmxwt)&(ii$bmxwt<=40),]
> nrow(ss)
[1] 10
```

A third distinction between `subset()` and indexing rows by logical vectors is that with `subset()`, we can omit dollar-sign notation if the data (the first argument) is a data frame. A subtle detail regarding this point is that `ee` is not actually a data frame:

```
> class(ee)
[1] "matrix"
```

So, the following results in an error:

```
> subset(ee,Estimate>2)
```

An easy way around this error is to make `ee` a data frame:

```
> ee<-as.data.frame(ee)
> subset(ee,Estimate>2)
             Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 62.677125  0.9289070 67.474055 0.000000e+00
genderMale   2.539453  0.3578733  7.095955 1.462151e-12
```

For the column names with spaces or other special characters we can add back quotes:

```
> subset(ee,`Pr(>|t|)`<=0.05)
              Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 62.6771255  0.9289070 67.474055 0.000000e+00
bmxbmi       0.2172014  0.0258695  8.396042 5.931366e-17
genderMale   2.5394527  0.3578733  7.095955 1.462151e-12
```

Indexing of columns

To keep some columns, the index must be placed *after the comma* because the space after the comma represents columns.

For example, from slide 53 of Module 4:

```
> ee
                Estimate Std. Error    t value     Pr(>|t|)
(Intercept) 62.677125492 0.92890704 67.4740555 0.000000e+00
```

```
bmxbmi       0.217201438 0.02586950  8.3960418 5.931366e-17
ridageyr    -0.005565416 0.01022388 -0.5443544 5.862216e-01
genderMale   2.539452656 0.35787328  7.0959549 1.462151e-12
> ee[,c(TRUE,FALSE,FALSE,TRUE)]
                Estimate     Pr(>|t|)
(Intercept) 62.677125492 0.000000e+00
bmxbmi       0.217201438 5.931366e-17
ridageyr    -0.005565416 5.862216e-01
genderMale   2.539452656 1.462151e-12
```

Here, are we indexing rows using a **logical vector**. Let's take a look at that logical vector:

```
> c(TRUE,FALSE,FALSE,TRUE)
[1]  TRUE FALSE FALSE  TRUE
```

Placing the logical vector after the comma results in subsetting of columns: we only get the columns of `ee` that are in the TRUE position.

Depending only on our goals and imagination, we can put *almost any* logical vector before the comma, so long as the logical vector's length matches the number of rows (this length requirement is not strictly true, but recommended for clarity).

Alternatively, we can index rows using a **numeric vector**. For example, from slide 66 of Module 4::

```
> ee[,c(1,4)]
                Estimate     Pr(>|t|)
(Intercept) 62.677125492 0.000000e+00
bmxbmi       0.217201438 5.931366e-17
ridageyr    -0.005565416 5.862216e-01
genderMale   2.539452656 1.462151e-12
```

Placing the numeric vector `c(1,4)` after the comma results in subsetting of columns: we only get the columns of `ee` that are in the numeric position `1` or `4`.

Depending only on our goals and imagination, we can put *almost any* numeric vector after the comma.

Finally, we can index rows using a **character vector**. For example:

```
> ee[,c('Estimate','Std. Error')]
                Estimate Std. Error
(Intercept) 62.677125492 0.92890704
bmxbmi       0.217201438 0.02586950
ridageyr    -0.005565416 0.01022388
genderMale   2.539452656 0.35787328
```

Placing the character vector `c('ridageyr','bmxbmi')` after the comma results in subsetting of columns: we only get the columns of `ee` named `Estimate` or `Std. Error`.

Indexing of rows and columns

We can index both rows and columns. For example, from slide 77 of Module 4:

```
> ii[1:5,c('bmxwt','bmxht')]
  bmxwt bmxht
1  78.3 171.3
2  89.5 176.8
3  88.9 175.3
```

```
4  52.0 162.4
5 105.0 158.7
```

Here, we are indexing rows using a numeric vector and indexing columns using a character vector.

The `data` argument

Many R functions support a `data` argument. For example, from slide 71 of Module 3:

```
> lm(bpxdi1~bmxbmi,data=ii)

Call:
lm(formula = bpxdi1 ~ bmxbmi, data = ii)

Coefficients:
(Intercept)       bmxbmi
    64.0737       0.2022
```

Similarly, an alternate version of the example on slide 18 of Module 3 is:

```
> t.test(lbdldl~hs,data=ii)

    Welch Two Sample t-test

data:  lbdldl by hs
t = 0.63877, df = 2329.3, p-value = 0.523
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1.878154  3.692851
sample estimates:
     mean in group No mean in group Smoking
            111.3549              110.4476
```

The `data` argument can be very useful, for two reasons. First, it allows us to avoid using dollar-sign notation. Second, *in combination with any of the subsetting techniques summarized above, it can be used to perform stratified analysis*. For example, to perform the t-test among individuals 25 years of age or younger:

```
> t.test(lbdldl~hs,data=ii[ii$ridageyr<=25,])

    Welch Two Sample t-test

data:  lbdldl by hs
t = 0.85901, df = 151.09, p-value = 0.3917
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -4.78518 12.14650
sample estimates:
     mean in group No mean in group Smoking
            96.90566              93.22500
```

Equivalently, using `subset()`:

```
> t.test(lbdldl~hs,data=subset(ii,ridageyr<=25))

    Welch Two Sample t-test
```

```
data:  lbdldl by hs
t = 0.85901, df = 151.09, p-value = 0.3917
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -4.78518 12.14650
sample estimates:
     mean in group No mean in group Smoking
             96.90566              93.22500
```

Functions that support the `data` argument include `plot()`, `t.test()`, `wilcox.test()`, `aov()`, `kruskal.test()`, `lm()`, and `ggplot()`.

The `with()` function

Not all R functions support the `data` argument. For example, the following example results in an error because the `table()` function does not have a `data` argument.

```
> table(hs,military,data=ii)
```

Fortunately, for R functions that do not have a `data` argument, a helpful alternative is to use the `with()` function. For example:

```
> with(ii,table(hs,military))
         military
hs        Military   No
  No           203 3033
  Smoking      339 2191
```

The first argument to the `with()` function is the data one wishes to use while the second argument is any arbitrary R expression involving that data. For example, from slide 36 of Module 4:

```
> ii$agegroup3<-with(ii,(ridageyr>=20)+(ridageyr>=40)+(ridageyr>=60))
> table(ii$agegroup3)

   1    2    3
1954 1974 1841
```

As with the `data` argument, the advantage of using `with()` is not only that it allows us to avoid dollar-sign notation, but also that *in combination with any of the subsetting techniques summarized above, it can be used to perform stratified analysis.* For example, to tabulate the two variables among individuals 40 years of age or younger:

```
> with(ii[ii$ridageyr<=40,],table(hs,military))
         military
hs        Military   No
  No           30 1253
  Smoking      30  748
```

Equivalently, using `subset()`:

```
> with(subset(ii,ridageyr<=40),table(hs,military))
         military
hs        Military   No
  No           30 1253
  Smoking      30  748
```

We can extend the complexity of the expression as we wish. For example:

```
> with(subset(ii,ridageyr<=40),chisq.test(table(hs,military)))

    Pearson's Chi-squared test with Yates' continuity correction

data:  table(hs, military)
X-squared = 3.4286, df = 1, p-value = 0.06408
```