Summarizing and managing data

Transforming a data frame into the format needed for analysis can be difficult! We have seen same basic methods already such as:

- Adding variables to a data frame using `data.frame()` or `set1$variable.name <- ____` like assignments
- Conditional functions like `ifelse()`
- Combine together data frames, vectors, or matrices through `rbind()` and `cbind()`
- Recycling

It can also be difficult to obtain intermediate summaries of data needed for a further analysis. The purpose of this section is to describe some additional tools available for summarizing and managing data. The programs used in this section are cereal_summary.R and data_management.R.

Base package

The merging of two data frames by a shared variable is a common task. Below is a simple example with the help of `merge()`.

```
> set1 <- data.frame(name1 = c("a", "b", "c", "d", "e", "f"), response1 = c(1,
    2, 3, 4, 5, 6))
> set2 <- data.frame(name2 = c("a", "a", "b", "c", "d", "e"), response2 = c(10,
    11, 20, 30, 40, 50))
> set1
  name1 response1
1     a         1
2     b         2
3     c         3
4     d         4
5     e         5
```

```
6     f         6
> set2
  name2 response2
1     a        10
2     a        11
3     b        20
4     c        30
5     d        40
6     e        50
> merge(x = set1, y = set2, by.x = "name1", by.y = "name2", all = TRUE)
  name1 response1 response2
1     a         1        10
2     a         1        11
3     b         2        20
4     c         3        30
5     d         4        40
6     e         5        50
7     f         6        NA
> merge(x = set1, y = set2, by.x = "name1", by.y = "name2", all = FALSE)
  name1 response1 response2
1     a         1        10
2     a         1        11
3     b         2        20
4     c         3        30
5     d         4        40
6     e         5        50
```

Finding all possible combinations of a number of categorical variables is greatly helped by the `expand.grid()` function:

```
> x <- 1:3
> y <- c("a", "b")
> expand.grid(x, y)
  Var1 Var2
1    1    a
2    2    a
3    3    a
4    1    b
5    2    b
```

```
6    3    b
> expand.grid(y, x)
  Var1 Var2
1    a    1
2    b    1
3    a    2
4    b    2
5    a    3
6    b    3
```

Notice that the last variable given in `expand.grid()` varies levels the slowest across the rows of the new data frame.

The sorting of a data frame is accomplished by `sort()` and `order()`:

```
> # Example 1
> x <- c("b", "c", 1)
> x
[1] "b" "c" "1"
> sort(x)
[1] "1" "b" "c"
>
> # Example 2
> set1 <- data.frame(ID = c(3, 1, 2), response = c(10, 20, 15))
> set1
  ID response
1  3       10
2  1       20
3  2       15
> sort(set1)  # Does not work
Error:  undefined columns selected
> order(set1$ID)
[1] 2 3 1
> set1[order(set1$ID), ]
  ID response
2  1       20
3  2       15
1  3       10
```

```
>
> # Example 3
> set1 <- data.frame(ID = c(2, 2, 1), response1 = c(20, 10, 15),
     response2 = c(20, 40, 18))
> set1
  ID response1 response2
1  2        20        20
2  2        10        40
3  1        15        18
> set1[order(set1$ID), ]
  ID response1 response2
3  1        15        18
1  2        20        20
2  2        10        40
> set1[order(set1$ID, set1$response1), ]
  ID response1 response2
3  1        15        18
2  2        10        40
1  2        20        20
```

Notice that order gives the row indices of the sorted data frame. Using these row indices with the data frame leads to the sorted data frame itself.

In the previous example, we also see *row names* given with each data frame. In the past, these usually made sense with a simple $1, ..., n$, labeling where n was the sample size. Due to the sorting, the row names do not make as much sense. To return these row names to a default $1, ..., n$ format, we can simply use the `row.names()` function:

```
> set1 <- data.frame(ID = c(2, 2, 1), response1 = c(20, 10, 15),
     response2 = c(20, 40, 18))
> set1
  ID response1 response2
1  2        20        20
2  2        10        40
3  1        15        18
```

```
> set2 <- set1[order(set1$ID), ]
> row.names(set2) <- NULL
> set2
  ID response1 response2
1  1        15        18
2  2        20        20
3  2        10        40
```

Changing the row names is often a common task. This occurs because one may work with an object to perform some calculations and the row names get carried forward to a new object. For this new object, these old row names may not be meaningful anymore. By using `row.names()` with the `NULL` value, the row names get reset to the default $1, \ldots, n$ format. Also, one could use other types of names with `row.names()` by combining them into a vector with `c()`. While less commonly done, the `names()` function can be used to rename the columns in a data frame.

The `reshape()` function can be quite useful for transforming a longitudinal data set from a "long" to a "wide" format and vice versa.

```
> # Wide format
> set1 <- data.frame(ID.name = c("subject1", "subject2", "subject3"),
     ID.number = c(1, 2, 3), age = c(19, 16, 21), response1 = c(1,
         0, 0), response2 = c(0, 0, 1))
> set1
   ID.name ID.number age response1 response2
1 subject1         1  19         1         0
2 subject2         2  16         0         0
3 subject3         3  21         0         1
> # Long format
> set2 <- reshape(data = set1, idvar = "ID.name", varying = c("response1",
     "response2"), v.names = "response", direction = "long", drop = "ID.number"
> set2
             ID.name age time response
subject1.1 subject1  19    1        1
subject2.1 subject2  16    1        0
```

```
subject3.1 subject3  21    1        0
subject1.2 subject1  19    2        0
subject2.2 subject2  16    2        0
subject3.2 subject3  21    2        1
> row.names(set2) <- NULL
> set2
   ID.name age time response
1 subject1  19    1        1
2 subject2  16    1        0
3 subject3  21    1        0
4 subject1  19    2        0
5 subject2  16    2        0
6 subject3  21    2        1
> #' Could also include age in idvar argument
> # reshape(data = set1, idvar = c('ID.name', 'age'), varying =
> # c('response1', 'response2'), v.names = 'response',
> # direction = 'long', drop = 'ID.number')
>
> # Back to wide format
> set3 <- reshape(data = set2, timevar = "time", idvar = "ID.name",
     direction = "wide", v.names = "response", sep = "")
> set3
   ID.name age response1 response2
1 subject1  19         1         0
2 subject2  16         0         0
3 subject3  21         0         1
```

We will often want to summarize a data set by a particular "grouping" variable. The `aggregate()` function does this by separating out the data by the grouping variable, applying a summary function to each data group, and then combining the summarized data back into a data frame. Below are a few examples for how to use `aggregate()` with the cereal data:

```
> # Location is for my computer
> cereal <- read.csv(file = "C:\\data\\cereal.csv")
> head(cereal, n = 3)
  ID Shelf                                    Cereal size_g sugar_g fat_g
```

```
1  1     1 Kellog's Razzle Dazzle Rice Crispies     28       10      0
2  2     1            Post Toasties Corn Flakes     28        2      0
3  3     1                Kellogg's Corn Flakes     28        2      0
  sodium_mg
1       170
2       270
3       300
> cereal$sugar <- cereal$sugar_g/cereal$size_g
> cereal$fat <- cereal$fat_g/cereal$size_g
> cereal$sodium <- cereal$sodium_mg/cereal$size_g
> # Summarize one variable
> aggregate(formula = sugar ~ Shelf, data = cereal, FUN = mean)
  Shelf  sugar
1     1 0.2568
2     2 0.4150
3     3 0.2304
4     4 0.2555
> # Summarize more than one variable
> aggregate(formula = cbind(sugar, sodium, fat) ~ Shelf, data = cereal,
     FUN = mean)
  Shelf  sugar sodium     fat
1     1 0.2568  8.039 0.02612
2     2 0.4150  5.273 0.04482
3     3 0.2304  4.461 0.02961
4     4 0.2555  4.686 0.02817
> # Summarize with a user created function that returns more
> # than one value
> mean.sd <- function(x) {
     c(mean(x), sd(x))
 }
> save.summary <- aggregate(formula = cbind(sugar, sodium, fat) ~
     Shelf, data = cereal, FUN = mean.sd)
> save.summary
  Shelf sugar.1 sugar.2 sodium.1 sodium.2   fat.1   fat.2
1     1 0.25684 0.16730    8.039    1.667 0.02612 0.03358
2     2 0.41497 0.09001    5.273    1.746 0.04482 0.02714
3     3 0.23037 0.15770    4.461    2.886 0.02961 0.02891
4     4 0.25548 0.11010    4.686    1.739 0.02817 0.01944
> names(save.summary)
```

```
[1] "Shelf"  "sugar"  "sodium" "fat"
> save.summary$sugar
       [,1]    [,2]
[1,] 0.2568 0.16730
[2,] 0.4150 0.09001
[3,] 0.2304 0.15770
[4,] 0.2555 0.11010
```

If the grouping mechanism involves more than one variable, one can use `var1 + var2` on the right side of the tilde in the `formula` argument.

Finding a subset of a data frame is often useful when we only want to examine a portion of a data set. We have used conditional arguments in the past to find these subsets. Another way to find a subset is through the `subset()` function:

```
> # Just shelf #1
> cereal[cereal$Shelf == 1, 8:10]
     sugar     fat sodium
1  0.35714 0.00000  6.071
2  0.07143 0.00000  9.643
3  0.07143 0.00000 10.714
4  0.06250 0.06250  8.750
5  0.43333 0.03333  7.000
6  0.35484 0.00000  5.806
7  0.44444 0.05556  7.407
8  0.33333 0.09259  7.407
9  0.37931 0.01724  7.586
10 0.06061 0.00000 10.000
> cereal[cereal$Shelf == 1, c("sugar", "fat", "sodium")]
     sugar     fat sodium
1  0.35714 0.00000  6.071
2  0.07143 0.00000  9.643
3  0.07143 0.00000 10.714
4  0.06250 0.06250  8.750
5  0.43333 0.03333  7.000
6  0.35484 0.00000  5.806
7  0.44444 0.05556  7.407
```

```
8  0.33333 0.09259  7.407
9  0.37931 0.01724  7.586
10 0.06061 0.00000 10.000
> subset(x = cereal, subset = Shelf == 1, select = c(sugar, fat,
     sodium))
     sugar     fat sodium
1  0.35714 0.00000  6.071
2  0.07143 0.00000  9.643
3  0.07143 0.00000 10.714
4  0.06250 0.06250  8.750
5  0.43333 0.03333  7.000
6  0.35484 0.00000  5.806
7  0.44444 0.05556  7.407
8  0.33333 0.09259  7.407
9  0.37931 0.01724  7.586
10 0.06061 0.00000 10.000
> # Just observations corresponding to the minimum of sodium
> cereal[cereal$sodium == min(cereal$sodium), 8:10]
   sugar    fat sodium
26  0.00 0.0102      0
30  0.02 0.0200      0
> subset(x = cereal, subset = sodium == min(sodium), select = c(sugar,
     fat, sodium))
   sugar    fat sodium
26  0.00 0.0102      0
30  0.02 0.0200      0
```

Perhaps the main advantage here is that the `subsets()` function makes the code a little more readable.

`plyr` and `reshape2` packages

Hadley Wickham provides similar functions to the above in his `plyr` and `reshape2` packages. These packages are quite popular, like his `ggplot2` package. There are numerous introductions to these packages available on the Internet. Other introductions can be found on Wickham's website[1], in his papers[2], and his ggplot2 book. Next, I will provide a brief introduction to these packages.

The plyr package is based on the idea that one often wants to break up a data structure, apply some function to each part, and then put back together the results into a new data structure, which is very similar to what aggregate() provides. The name "plyr" comes about because functions within the package act somewhat like the apply() function that we saw earlier (the "ply" part comes from ap"ply" and the r is for R). Many of the main functions in plyr are named using a particular convention:

- First letter denotes the type of data object to include in the .data argument
- Second letter denotes the type of data object to put the resulting calculations into
- End with "ply"

Below are examples using ddplyr():

```
> library(package = "plyr")
> # Example #1 - Using subset() from base package
> ddply(.data = cereal, .variables = "Shelf", .fun = subset,
    select = c("sugar", "fat", "sodium"), sodium ==
        min(sodium))
  Shelf  sugar     fat sodium
1     1 0.3548 0.00000  5.806
2     2 0.5556 0.01852  1.852
3     3 0.0000 0.01020  0.000
4     3 0.0200 0.02000  0.000
5     4 0.2545 0.05455  1.818
> # Example #2: Using summarize() from dplyr package
> ddply(.data = cereal, .variables = "Shelf", .fun = summarize,
    mean.sugar = mean(sugar), mean.sodium = mean(sodium),
    sd.sodium = sd(sodium))
```

[1] http://plyr.had.co.nz

[2] https://www.jstatsoft.org/article/view/v040i01 and https://www.jstatsoft.org/article/view/v040i01

```
  Shelf mean.sugar mean.sodium sd.sodium
1     1     0.2568       8.039     1.667
2     2     0.4150       5.273     1.746
3     3     0.2304       4.461     2.886
4     4     0.2555       4.686     1.739
> # Example #3: Using transform() from base package
> # Standardize over all observations
> head(scale(cereal$sodium), n = 2)
       [,1]
[1,] 0.1855
[2,] 1.6358
> # Standardize for Shelf = 1
> head(scale(cereal$sodium[cereal$Shelf == 1]), n = 2)
        [,1]
[1,] -1.1799
[2,]  0.9622
> save.res <- ddply(.data = cereal, .variables = "Shelf",
     .fun = transform, sodium.z = scale(sodium))
> head(save.res, n = 2)
  ID Shelf                               Cereal size_g sugar_g fat_g
1  1     1 Kellog's Razzle Dazzle Rice Crispies     28      10     0
2  2     1            Post Toasties Corn Flakes     28       2     0
  sodium_mg   sugar fat sodium sodium.z
1       170 0.35714   0  6.071  -1.1799
2       270 0.07143   0  9.643   0.9622
```

Comments:

- Example #1 gives all variables in the data frame corresponding to the minimum sodium levels for each shelf. Note that there were two cereals tied for the minimum in shelf #3. The `.variables` argument gives the group variable for the data. If there was more than one variable, one could use `c()` to combine the variable names. Also, equivalent forms of syntax for `.variables = "Shelf"` are `.variables = .(Shelf)` and `.variables = ~ Shelf`.

- Example #2 uses a function from `plyr` called `summarize()`.

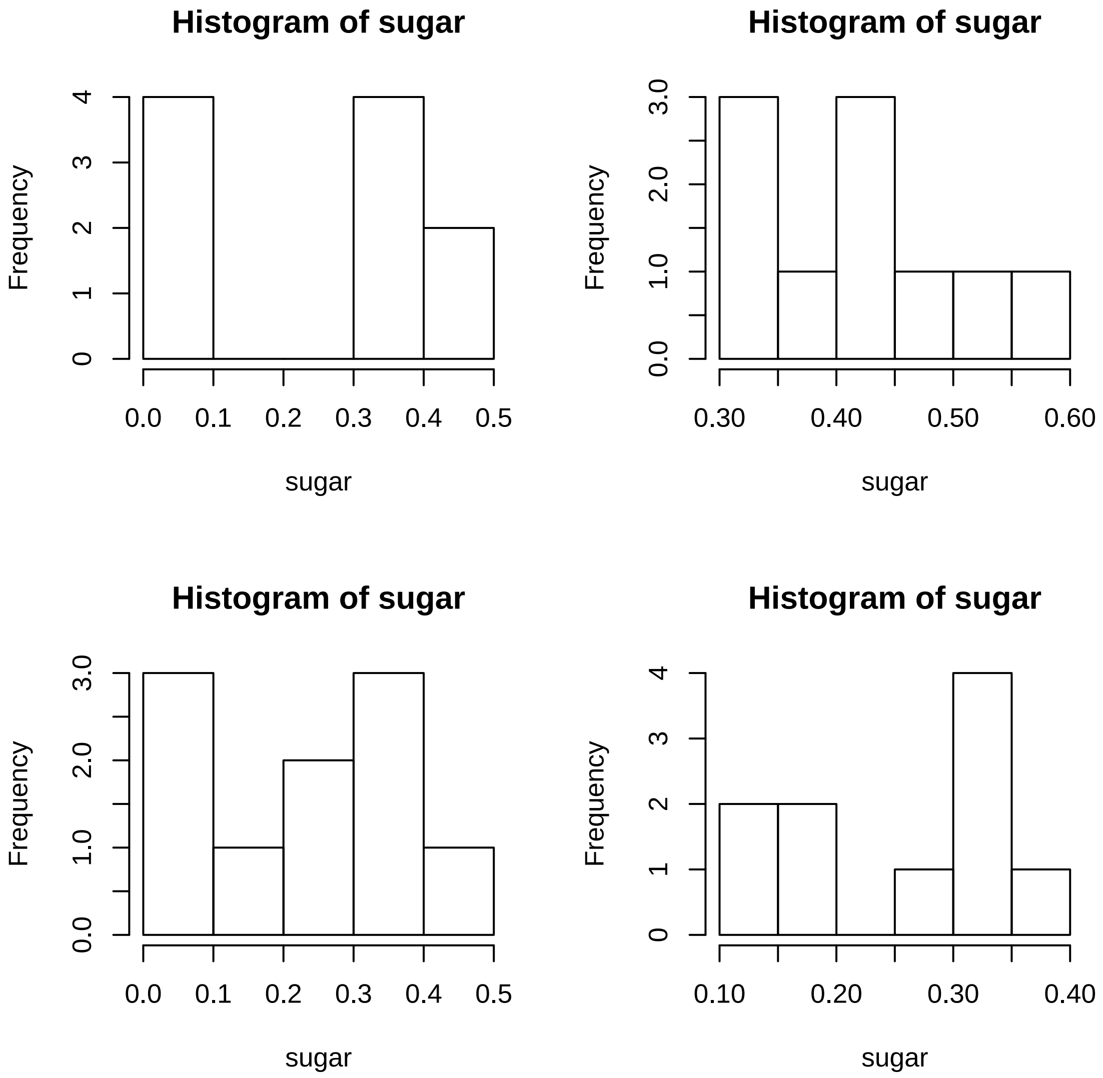
This allows one to apply a function that returns a computed result. My program contains an example of where I wrote my own function to use with the `.fun` argument.

- Example #3 shows how to transform each value for a variable to a new value. For the example here, I use the `scale()` function from the `base` package to standardize observations (i.e., $z = \frac{y-\bar{y}}{s}$ where $\bar{y}$ is the sample mean and s is the sample standard deviation).

Below are a few additional examples.

```
> # dlply() - Creates a list
> save.list <- dlply(.data = cereal, .variables = "Shelf", .fun = summarize,
     mean.sugar = mean(sugar))
> names(save.list)
[1] "1" "2" "3" "4"
> save.list
$`1`
  mean.sugar
1     0.2568

$`2`
  mean.sugar
1      0.415

$`3`
  mean.sugar
1     0.2304

$`4`
  mean.sugar
1     0.2555

attr(,"split_type")
[1] "data.frame"
attr(,"split_labels")
  Shelf
1     1
```

```
2     2
3     3
4     4
> save.list$"1"
  mean.sugar
1     0.2568
> #' d_ply - Underscore is used for plotting - output is discarded
> #'  Notice that the x-axis scales are not the same
> par(mfrow = c(2, 2))
> d_ply(.data = cereal, .variables = "Shelf", .fun = transform,
     hist(sugar))
```



```
> par(mfrow = c(1, 1))
```

The `reshape2` package provides similar functionality to the reshape() function from the base package. There are two main functions in it: melt(), "wide" to a "long" format, and dcast(), "long" to "wide" format for data frames.

```
> library(package = "reshape2")
> # Wide format from earlier
> set1
   ID.name ID.number age response1 response2
1 subject1         1  19         1         0
2 subject2         2  16         0         0
3 subject3         3  21         0         1
> # Convert to long format - need to include age in
> # id.vars otherwise will not be included in data
> # frame
> set2 <- melt(data = set1, id.vars = c("ID.name", "age"),
    measure.vars = c("response1", "response2"), variable.name = "time",
    value.name = "response")
> set2
   ID.name age      time response
1 subject1  19 response1        1
2 subject2  16 response1        0
3 subject3  21 response1        0
4 subject1  19 response2        0
5 subject2  16 response2        0
6 subject3  21 response2        1
> # Back to wide format
> set3 <- dcast(data = set2, formula = ID.name + age ~
    time, value.var = "response")
> set3
   ID.name age response1 response2
1 subject1  19         1         0
2 subject2  16         0         0
3 subject3  21         0         1
```

SQL

A general purpose language/syntax for working with databases is the Structured Query Language (SQL). This language did not come from statisticians so it has quite a different syntax to it than what we have seen before. For those individuals used to the SQL environment, R packages have been written so that one can used the same type of syntax in R. These packages include `sqldf`, `RSQLite`, and `RMySQL`. See `http://www.r-bloggers.com/make-r-speak-sql-with-sqldf` and `https://www.simple-talk.com/dotnet/software-tools/sql-and-r-` for some background information about SQL and R.