LeaRning Week 4: Reshaping, melting and casting

LeaRning Team

19/07/2021

```
##
##  --------------
## How about them bar graphs?
##  --------------
##     \
##       \
##         \
##           _____
##        .'` ,-. `'.
##       /   ([ ])   \
##      /.-""`(`)`""-.\
##       <'```(.)```'>
##       <'```(.)```'>
##        <'``(.)``'>
##    sk   <``\_/``>
##          `'___'`
##
```

Let's load the packages that we will need for this week's lesson, then import our data.

```r
library(plyr)
library(tidyverse)
library(reshape2)
```

We will be working with two data sets. The first, which we will call "dat", is our experimental data. The 2nd, which we will call "meta", contains our independent variables.

```r
dat <- read.csv("Week4_merging_reshaping/LeaRning_week4_data_matrix.csv")
meta <- read.csv("Week4_merging_reshaping/LeaRning_week4_metadata_raw.csv")
```

Have a quick view of these data sets to get a hang of their structure.

```r
head(dat)
```

```
##   sample.id Well IFN.a IFN.g IL.10  IL.23   IL.6 MIP.1b TNF.a
## 1   sample1   D3 21.24 18.51 24.85 118.22 753.85 338.80 73.68
## 2  sample10   E5 21.24  3.05  5.27 118.22  56.82  49.76  9.75
## 3  sample11   F5 21.24  3.05  5.27 118.22  35.69  52.08  9.75
## 4  sample12   G5 21.24  3.05  5.27 118.22  58.38  67.67  9.75
## 5  sample13   A6 21.24  3.10 21.49 118.22 355.29 151.20 41.90
## 6  sample14   B6 21.24  3.05 15.67 118.22 186.43 217.70 49.68
```

```r
head(meta)
```

```
##   sample.id MouseID          Group Tissue
## 1   SAMPLE1    trt2  treatment_LPS SPLEEN
## 2   sample2    trt3  treatment_LPS SPLEEN
## 3   sample3    trt4  treatment_LPS SPLEEN
## 4   sample4    trt5  treatment_LPS SPLEEN
## 5   sample5    trt2 treatment_R848 SPLEEN
## 6   sample6    trt3 treatment_R848 SPLEEN
```

Reshaping the data

First, we need to melt the data so that all of the variables (cytokines) are in one column.
i.e. we need to make the data long.

We use the reshape2 package for this. It has two main functions: melt and cast.
When we cast data frames, the function becomes dcast.
First, we need to melt the data.

```r
?melt()
```

melt() takes three main arguments: The data, the identifier variables -> id.vars (i.e. the independent variables or sample descriptors) and the measured variables -> measure.vars (dependent variables).

We can also specify what we want the name of the variable column and the value column to be.

Step 1: Define the id.variables

```r
colnames(dat)
```

```
## [1] "sample.id" "Well"      "IFN.a"     "IFN.g"     "IL.10"     "IL.23"
## [7] "IL.6"      "MIP.1b"    "TNF.a"
```

Here, the identifier variables are the sample ID (sample.id) and Well. All the other variables are the independent measured variables. Let's create two vectors to separate this information.

```r
id.v <- colnames(dat)[1:2]
id.v
```

```
## [1] "sample.id" "Well"
```

```r
m.v <- colnames(dat)[3:ncol(dat)]
m.v
```

```
## [1] "IFN.a"  "IFN.g"  "IL.10"  "IL.23"  "IL.6"   "MIP.1b" "TNF.a"
```

Step 2: melt the data into long format

Now we are ready to melt our data. We want all our measure variables in one column, called "variable" and the corresponding values in a separate column. Below, we will first do a simple melt, and then add a bit of code to make the column names more intuitive for our later use.

```r
dat.m <- melt(dat, id.vars = id.v, measure.vars = m.v)
# Have a look at the "melted" data:

head(dat.m)
```

```
##   sample.id Well variable value
## 1   sample1   D3    IFN.a 21.24
## 2  sample10   E5    IFN.a 21.24
## 3  sample11   F5    IFN.a 21.24
## 4  sample12   G5    IFN.a 21.24
## 5  sample13   A6    IFN.a 21.24
## 6  sample14   B6    IFN.a 21.24
```

```r
# Option to re-name the columns "value" and "variable"
dat.m <- melt(dat, id.vars = id.v, measure.vars = m.v, value.name = "pg/mL", variable.name = "Cytokine"
head(dat.m)
```

```
##   sample.id Well Cytokine pg/mL
## 1   sample1   D3    IFN.a 21.24
## 2  sample10   E5    IFN.a 21.24
## 3  sample11   F5    IFN.a 21.24
## 4  sample12   G5    IFN.a 21.24
## 5  sample13   A6    IFN.a 21.24
## 6  sample14   B6    IFN.a 21.24
```

Step 3: cast the data back to wide-format

We need to provide the function dcast() with a formula to describe the shape of the data.

The variables on the left of the formula are the ID variables, and those on the right are the measured variables

```r
dat.c <- dcast(dat.m, sample.id + Well ~ Cytokine, value.var = "pg/mL")

head(dat.c)
```

```
##   sample.id Well IFN.a IFN.g IL.10  IL.23   IL.6 MIP.1b TNF.a
## 1   sample1   D3 21.24 18.51 24.85 118.22 753.85 338.80 73.68
## 2  sample10   E5 21.24  3.05  5.27 118.22  56.82  49.76  9.75
## 3  sample11   F5 21.24  3.05  5.27 118.22  35.69  52.08  9.75
## 4  sample12   G5 21.24  3.05  5.27 118.22  58.38  67.67  9.75
## 5  sample13   A6 21.24  3.10 21.49 118.22 355.29 151.20 41.90
## 6  sample14   B6 21.24  3.05 15.67 118.22 186.43 217.70 49.68
```

Alternative method using tidyr - a tidyverse package

We can use the function pivot_longer() to achieve the same thing as melt().

```r
?pivot_longer()
```

```r
dat.long <- pivot_longer(dat, cols = !c(sample.id, Well), names_to = "Cytokine", values_to = "value")

head(dat.long)
```

```
## # A tibble: 6 x 4
##   sample.id Well  Cytokine value
##   <fct>     <fct> <chr>    <dbl>
## 1 sample1   D3    IFN.a     21.2
## 2 sample1   D3    IFN.g     18.5
## 3 sample1   D3    IL.10     24.8
## 4 sample1   D3    IL.23    118.
## 5 sample1   D3    IL.6     754.
## 6 sample1   D3    MIP.1b   339.
```

Similarly, we can use pivot_wider() to achieve the same thing as dcast().

```r
dat.wide <- pivot_wider(dat.long, names_from = Cytokine, values_from = "value")

head(dat.wide)
```

```
## # A tibble: 6 x 9
##   sample.id Well  IFN.a IFN.g IL.10 IL.23  IL.6 MIP.1b TNF.a
##   <fct>     <fct> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>
## 1 sample1   D3     21.2 18.5  24.8   118. 754.   339.  73.7
## 2 sample10  E5     21.2  3.05  5.27  118.  56.8   49.8  9.75
## 3 sample11  F5     21.2  3.05  5.27  118.  35.7   52.1  9.75
## 4 sample12  G5     21.2  3.05  5.27  118.  58.4   67.7  9.75
## 5 sample13  A6     21.2  3.1  21.5   118. 355.   151.  41.9
## 6 sample14  B6     21.2  3.05 15.7   118. 186.   218.  49.7
```

Joining the data

There are various ways to join data sets using R, depending on what your goals are.
While the code is straightforward, there are a few small things to watch out for to ensure you get the result you were hoping for.

Here, we will go through three ways of joining two data sets together:

1. Binding rows or columns with rbind() and cbind() 2. Using merge() from base R 3. join() from the plyr package.

cbind() and rbind()

These two functions are quite simple. cbind() takes columns from two data sets with equivalent row names, and pastes them together.
rbind() does the equivalent for rows. This is handy, but you must have to data sets with identical row or column names. Let's have a quick look at the usage from our data matrix.

cbind()

First, let's assign row names to our data.

```r
rownames(dat.m) <- paste0("R", 1:nrow(dat.m))
```

Now, let's make two data sets to join

```r
c1 <- dat.m[,c(1:2)]
c2 <- dat.m[,c(3:4)]
```

Now, let's join them!

```r
c.both <- cbind(c1, c2)
```

rbind()

Since we already have column names, we just need to split up our data.

```r
r1 <- dat.m[c(1:5),]
r2 <- dat.m[c(6:10),]
```

Let's quickly change a column name to see what happens

```r
colnames(r1)[1] <- toupper(colnames(r1)[1])

r.both <- rbind(r1, r2)
```

```
## Error in match.names(clabs, names(xi)): names do not match previous names
```

We get an error. Rbind will first check for matching column names.
If there are any mismatches, then the function will not work. Let's fix this.

```r
colnames(r1)[1] <- tolower(colnames(r1)[1])

r.both <- rbind(r1, r2)
```

Great! That's the simplicity of rbind and cbind. But we know that most of the time, we need to combine different variables from different data sets where the column and row names are not necessarily identical. Let's start with the help function for join.

join()

the join function implemented through plyr gives us much more power, especially since often we do not have two data frames with equivalent information, the same columns, or the same rows. Join allows us to still combine these data similarly to VLOOKUP for excel users, but with greater accuracy and control.

```r
?join()
```

The basic arguments of join() are the two data frames, by (the variables you want to use to join by), type, and match.
'Type' refers to the way you want the data to be combined. A left join will retain all rows in x, and add the matching information from y. A right join will perform the opposite function, and the inner join will only retain information common between the two data sets.

'match' is an important one. If your 'by' variable can be matched by several rows, then the data join will retain the information from all rows. A right match returns all the rows in x, and a left match will match the rows in y. This is best illustrated using an example.

Let's join our metadata to our melted data frame dat.m, and save it to a new object called dat.meta. Saving to a new object is useful, as we can then use it to see if we got what we were expecting.

```
dat.meta <- join(dat.m,
                 meta,
                 by = "sample.id",
                 type = "left",
                 match = "all")
```

Now, if this worked, logically, dat.meta should have the same number of rows as dat.m.

```
nrow(dat.m)
```

```
## [1] 343
```

```
nrow(dat.meta)
```

```
## [1] 350
```

Clearly this is not the case!
We have picked up 7 extra rows during our merge. What might have happened?

Let's see if we can find the variable with extra values.

```
table(dat.m$sample.id)
```

```
##
##  sample1 sample10 sample11 sample12 sample13 sample14 sample15 sample16
##        7        7        7        7        7        7        7        7
## sample17 sample19  sample2 sample20 sample21 sample22 sample23 sample25
##        7        7        7        7        7        7        7        7
## sample26 sample27 sample28 sample29  sample3 sample30 sample31 sample32
##        7        7        7        7        7        7        7        7
## sample33 sample34 sample35 sample36 sample37 sample38 sample39  sample4
##        7        7        7        7        7        7        7        7
## sample40 sample41 sample42 sample43 sample44 sample45 sample46 sample47
##        7        7        7        7        7        7        7        7
## sample48 sample49  sample5 sample50 sample51  sample6  sample7  sample8
##        7        7        7        7        7        7        7        7
##  sample9
##        7
```

There are 7 rows for each sample ID.

```
table(dat.meta$sample.id)
```

```
##
##  sample1 sample10 sample11 sample12 sample13 sample14 sample15 sample16
##        7        7        7        7        7        7        7        7
## sample17 sample19  sample2 sample20 sample21 sample22 sample23 sample25
##        7        7        7        7        7        7        7        7
## sample26 sample27 sample28 sample29  sample3 sample30 sample31 sample32
##        7        7        7        7        7        7        7        7
## sample33 sample34 sample35 sample36 sample37 sample38 sample39  sample4
##        7        7        7        7        7        7        7        7
## sample40 sample41 sample42 sample43 sample44 sample45 sample46 sample47
##        7        7        7        7        7        7        7        7
## sample48 sample49  sample5 sample50 sample51  sample6  sample7  sample8
##        7        7        7        7       14        7        7        7
##  sample9
##        7
```

sample51 has 14 variables! Let's explore this.
One way to do this is to first look for duplicate values in your metadata and in your original data.
We can do this by combining the which() and duplicated() functions.

```
dup <- which(duplicated(dat$sample.id))
dup
```

```
## integer(0)
```

In our original data set, there are no duplicated sample IDs.

```
dup <- which(duplicated(meta$sample.id))
dup
```

```
## [1] 52
```

This returns the row coordinate of our duplicated sample ID. Let's get it.

```
id.dup <- meta$sample.id[dup]
```

Now let's have a look at all the duplicated rows.

```
dup.data <- meta[which(meta$sample.id %in% id.dup),]
dup.data
```

```
##    sample.id MouseID      Group Tissue
## 51  sample51    ctr5 control_US SPLEEN
## 52  sample51    <NA>       <NA>   <NA>
```

We can spot the issue. sample51 is duplicated, and in the second instance there are no values for any of the variables.
Now let's see how this was handled in the joined data.

```
dup.data.m <- dat.meta[which(dat.meta$sample.id %in% id.dup),]
dup.data.m
```

```
##     sample.id Well Cytokine  pg/mL MouseID      Group Tissue
## 45   sample51   H8    IFN.a  21.24    ctr5 control_US SPLEEN
## 46   sample51   H8    IFN.a  21.24    <NA>       <NA>   <NA>
## 95   sample51   H8    IFN.g   3.05    ctr5 control_US SPLEEN
## 96   sample51   H8    IFN.g   3.05    <NA>       <NA>   <NA>
## 145  sample51   H8    IL.10   5.27    ctr5 control_US SPLEEN
## 146  sample51   H8    IL.10   5.27    <NA>       <NA>   <NA>
## 195  sample51   H8    IL.23 118.22    ctr5 control_US SPLEEN
## 196  sample51   H8    IL.23 118.22    <NA>       <NA>   <NA>
## 245  sample51   H8     IL.6  52.07    ctr5 control_US SPLEEN
## 246  sample51   H8     IL.6  52.07    <NA>       <NA>   <NA>
## 295  sample51   H8   MIP.1b  89.33    ctr5 control_US SPLEEN
## 296  sample51   H8   MIP.1b  89.33    <NA>       <NA>   <NA>
## 345  sample51   H8    TNF.a   9.75    ctr5 control_US SPLEEN
## 346  sample51   H8    TNF.a   9.75    <NA>       <NA>   <NA>
```

Now we see that both rows were included in the join, even though we specified it to be a "left" join.
We can fix this in two ways:

1. We can specify "match" to be "first"
2. If we are not confident in this, the best thing to do is to clean up your metadata and then join it.

Let's try #1.

```
dat.meta <- join(dat.m,
                 meta,
                 by = "sample.id",
                 type = "left",
                 match = "first")
nrow(dat.m)
```

```
## [1] 343
```

```
nrow(dat.meta)
```

```
## [1] 343
```

```
# All seems to be in order.  Let's look at sample51 again.

dat.meta[which(dat.meta$sample.id == "sample51"),]
```

```
##      sample.id Well Cytokine  pg/mL MouseID      Group Tissue
## R45   sample51   H8    IFN.a  21.24    ctr5 control_US SPLEEN
## R94   sample51   H8    IFN.g   3.05    ctr5 control_US SPLEEN
## R143  sample51   H8    IL.10   5.27    ctr5 control_US SPLEEN
## R192  sample51   H8    IL.23 118.22    ctr5 control_US SPLEEN
## R241  sample51   H8     IL.6  52.07    ctr5 control_US SPLEEN
## R290  sample51   H8   MIP.1b  89.33    ctr5 control_US SPLEEN
## R339  sample51   H8    TNF.a   9.75    ctr5 control_US SPLEEN
```

Great! However, as you may have already concluded, if the "missing data" row came first, this would not have worked.
So let's go ahead and remove that second duplicate. And we can practice using our logical expressions while we're at it.

Logically, we can remove the duplicate we don't want by flagging the row where the sampleID is sample51 and there is a missing value in any other column, say MouseID.

```
meta.clean <- meta[-which(meta$sample.id == "sample51" & is.na(meta$MouseID)),]
```

Now let's try that join again.

```
dat.meta <- join(dat.m,
                 meta.clean,
                 by = "sample.id",
                 type = "left",
                 match = "all")

nrow(dat.meta)
```

```
## [1] 343
```

```
nrow(dat.m)
```

```
## [1] 343
```

The other thing that can go wrong is missing data in your join. This can happen one of two ways:
1. There are variables with missing data in the metadata, so when the data is joined there is nothing there to join.
2. There are some values in your anchor column that are absent in the metadata.

Let's check for missing data in the Group column. We can do this by combining the which() and is.na() functions, as we've used above.

```
which(is.na(meta.clean$Group))
```

```
## integer(0)
```

There is no missing data! Now, let's check our joined data.

```
which(is.na(dat.meta$Group))
```

```
## [1]   1  50  99 148 197 246 295
```

There are a few columns that have missing values in the group column. Let's check it out.

```
check.missing <- dat.meta[which(is.na(dat.meta$Group)),]
check.missing
```

```
##     sample.id Well Cytokine  pg/mL MouseID Group Tissue
## 1     sample1   D3    IFN.a  21.24    <NA>  <NA>   <NA>
## 50    sample1   D3    IFN.g  18.51    <NA>  <NA>   <NA>
## 99    sample1   D3    IL.10  24.85    <NA>  <NA>   <NA>
## 148   sample1   D3    IL.23 118.22    <NA>  <NA>   <NA>
## 197   sample1   D3     IL.6 753.85    <NA>  <NA>   <NA>
## 246   sample1   D3   MIP.1b 338.80    <NA>  <NA>   <NA>
## 295   sample1   D3    TNF.a  73.68    <NA>  <NA>   <NA>
```

These seem to all be coming from one sample. Now, there are a few ways to do this kind of detective work.

1. For the first way, we can search the metadata for sample1.
2. For the second way, we can get a list of all the differences between the sample IDs in both data sets (handy if you might have a few discrepancies).

Let's try the first way.

```
which(meta.clean$sample.id == "sample1")
```

```
## integer(0)
```

No sample1!

Let's try the 2nd way.
We can use the setdiff() function to find elements that do not intersect across two vectors.

```
diffs <- setdiff(dat.m$sample.id, meta.clean$sample.id)
diffs
```

```
## [1] "sample1"
```

This returned elements in x that are not in y. If we want it the other way around as well, then we can specify both of these as two lists.
Hang in there... it is not as cumbersome as it sounds.

```
diffs <- c(setdiff(dat.m$sample.id, meta.clean$sample.id),
           setdiff(meta.clean$sample.id, dat.m$sample.id))
diffs
```

```
## [1] "sample1"  "SAMPLE1"  "sample18" "sample24"
```

This was very informative! First, we found out that "sample1" is not in the metadata, but also that "SAMPLE1", "sample18", and "sample24" are in the metadata, but not in our data matrix.
Unless we have reason to suspect these were left out, we now know that there is just some extra information in the metadata.
However, we can easily spot the 2nd issue. sample1 was for some reason entered as uppercase.

Let's clean this up.

```
meta.clean$sample.id <- tolower(meta.clean$sample.id)

unique(meta.clean$sample.id)
```

```
##  [1] "sample1"  "sample2"  "sample3"  "sample4"  "sample5"  "sample6" 
##  [7] "sample7"  "sample8"  "sample9"  "sample10" "sample11" "sample12"
## [13] "sample13" "sample14" "sample15" "sample16" "sample17" "sample18"
## [19] "sample19" "sample20" "sample21" "sample22" "sample23" "sample24"
## [25] "sample25" "sample26" "sample27" "sample28" "sample29" "sample30"
## [31] "sample31" "sample32" "sample33" "sample34" "sample35" "sample36"
## [37] "sample37" "sample38" "sample39" "sample40" "sample41" "sample42"
## [43] "sample43" "sample44" "sample45" "sample46" "sample47" "sample48"
## [49] "sample49" "sample50" "sample51"
```

Seems like all is in order. Let's try that join once more!

```r
dat.meta <- join(dat.m,
                 meta.clean,
                 by = "sample.id",
                 type = "left",
                 match = "all")

nrow(dat.meta)
```

```
## [1] 343
```

```r
nrow(dat.m)
```

```
## [1] 343
```

And let's check for any missing data.

```r
which(is.na(dat.meta$Group))
```

```
## integer(0)
```

Great! We have successfully joined our data. Now, for this class, we worked a bit backwards.
Usually, the cleaning should come first. You can save yourself some detective work if you perform a few routine checks on your data first.
These include:

1. Check to see if all your anchor variables in data set x are represented in data set y. 2. Check for unexpected duplicates in both data sets.

Once your data passes both these tests, you are sure to get a successful join!

Summarizing the data

There are many tools we can use to summarize our data.

Some of them simply allow us to conduct 'checks' or observe our data without doing anything to it.
For example, the str() or summary() functions:

```r
# str() can be used to quickly glance at the nature of our data.
# Number of rows, columns, column names and type of elements they contain etc..
str(dat.meta)
```

```
## 'data.frame':    343 obs. of  7 variables:
##  $ sample.id: Factor w/ 49 levels "sample1","sample10",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ Well     : Factor w/ 49 levels "A10","A11","A6",..: 21 30 35 40 3 9 15 24 31 41 ...
##  $ Cytokine : Factor w/ 7 levels "IFN.a","IFN.g",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ pg/mL    : num  21.2 21.2 21.2 21.2 21.2 ...
##  $ MouseID  : Factor w/ 17 levels "ctr1","ctr2",..: 10 11 12 13 15 16 17 1 2 4 ...
##  $ Group    : Factor w/ 6 levels "control_LPS",..: 4 6 6 6 4 4 4 1 1 1 ...
##  $ Tissue   : Factor w/ 1 level "SPLEEN": 1 1 1 1 1 1 1 1 1 1 ...
```

```r
# summary() essentially does the same thing, albeit in a more elegant way.
# If summary() is used on a dataframe like below it will return basic information about it.
# However, summary() can be used for a variety of things, notably for statistical
# testing which we will explore further in the future.
summary(dat.meta)
```

```
##     sample.id        Well       Cytokine      pg/mL             MouseID
##  sample1 :  7   A10    :  7   IFN.a :49   Min.   :    3.05   ctr1   : 21
##  sample10:  7   A11    :  7   IFN.g :49   1st Qu.:   21.24   ctr4   : 21
##  sample11:  7   A6     :  7   IL.10 :49   Median :   74.98   ctr5   : 21
##  sample12:  7   A7     :  7   IL.23 :49   Mean   :  527.62   ctr6   : 21
##  sample13:  7   A8     :  7   IL.6  :49   3rd Qu.:  194.34   ctr7   : 21
##  sample14:  7   A9     :  7   MIP.1b:49   Max.   :13507.63   ctr8   : 21
##  (Other) :301   (Other):301   TNF.a :49                      (Other):217
##             Group       Tissue
##  control_LPS   :56   SPLEEN:343
##  control_R848  :56
##  control_US    :63
##  treatment_LPS :56
##  treatment_R848:56
##  treatment_US  :56
##
```

Some functions allow us to do work on our data to explore particular aspects of it or summarize it in a different way.

For example, I can grab the mean of the whole pg/ml column.

```r
mean(dat.meta$`pg/mL`)
```

```
## [1] 527.6194
```

But what if I wanted to find the mean pg/ml by Cytokine?

A quick glance at the 'apply' family of functions.

```r
# Have a look at what this returns.
tapply(dat.meta$`pg/mL`, dat.meta$Cytokine, mean)
```

```
##      IFN.a      IFN.g      IL.10      IL.23       IL.6     MIP.1b      TNF.a
##   25.91163  175.96204   60.67245  118.22000 2754.56306  480.11204   77.89429
```

```r
# Roughly, this is what tapply() just did:
# 1) Sort the 'pg/ml' column by its corresponding Cytokine.
# 2) Take the mean of all the 'pg/ml' values from each newly sorted 'group'.
# 3) Return a vector with the name of the Cytokine and calculated pg/ml mean as a value.
```

tapply() computes a measure (mean, median, min, max, etc..) or a function for each factor variable in a vector. It is a very useful function that lets you create a subset of a vector and then apply some functions to each of the subset.

tapply() is part of the 'apply' family of functions which includes apply(), sapply(), lapply() and tapply(). The 'apply' functions are essentially doing the same thing, but on different data types and each returning something different. In brief, they 'grab' the data that you give them, and 'apply' a function upon it, then return the result back to you. sapply() returns a vector, while lapply returns a list, for example.

These functions are quite powerful and we will be learning more about them in the future.

Great, we know the mean pg/ml for each Cytokine, which is probably not all useful to us at the moment. We want to get the mean for each cytokine, but for each treatment separately.

Just like there are functions to summarize parts of our dataset, there are some which can summarize it in a broader way.

Let's summarize!

Before proceeding, there is some hidden information in our current dataset which we can access.

```r
unique(dat.meta$Group)
```

```
## [1] treatment_LPS  treatment_US   control_LPS    treatment_R848 control_R848
## [6] control_US
## 6 Levels: control_LPS control_R848 control_US treatment_LPS ... treatment_US
```

We can see taht in our Group column there are treatment/control groups, as well as LPS/R848/US stimuli. This data set is breaking the "one type of information per variable" rule.

Let's separate those two variables so that we can access them during our analyses.
Here, we are using the separate() function from the tidyverse package (tidyr).

```r
dat.meta <- separate(dat.meta, Group, c("group", "stimulus"), sep = "_")
# Let's have a quick look
head(dat.meta[,6:7])
```

```
##       group stimulus
## 1 treatment      LPS
## 2 treatment       US
## 3 treatment       US
## 4 treatment       US
## 5 treatment      LPS
## 6 treatment      LPS
```

That's better! Now, let's see what the ddply() function can do for us in our quest for a good summary. ddply is a part of the "apply" family, implemented in the plyr package. ddply takes a data frame, performs a function, and returns another data frame. Other functions in this class include dlply (data frame to list), ldply (list to data frame), and llply (list to list). As a part of our summary, we want to calculate standard error. Unfortunately, base R does not have a function for it. But we can easily make our own!

```r
standard_error <- function(x) sd(x) / sqrt(length(x))
```

If you look over to your environment, you now have a new function available to you. Now let's put this function to work with other functions in base R to summarise our data.

```
dat.sum <- ddply(dat.meta,
                 .(Cytokine, group, stimulus),
                 summarise,
                 mean = mean(log10(1+`pg/mL`)),
                 st.err = standard_error(log10(1+`pg/mL`)))
head(dat.sum)
```

```
##   Cytokine     group stimulus     mean     st.err
## 1    IFN.a   control      LPS 1.347135 0.00000000
## 2    IFN.a   control     R848 1.609895 0.05978126
## 3    IFN.a   control       US 1.347135 0.00000000
## 4    IFN.a treatment      LPS 1.347135 0.00000000
## 5    IFN.a treatment     R848 1.456684 0.04399411
## 6    IFN.a treatment       US 1.347135 0.00000000
```

Let's break down what ddply() just did.
We asked it to take the dat.meta dataframe, and create subsets for each combinations of the variables we gave it: Cytokine, group, and stimulus.
For each subset, we pass the function summarise(), and tell it to generate a mean and st.err, just like the example below.

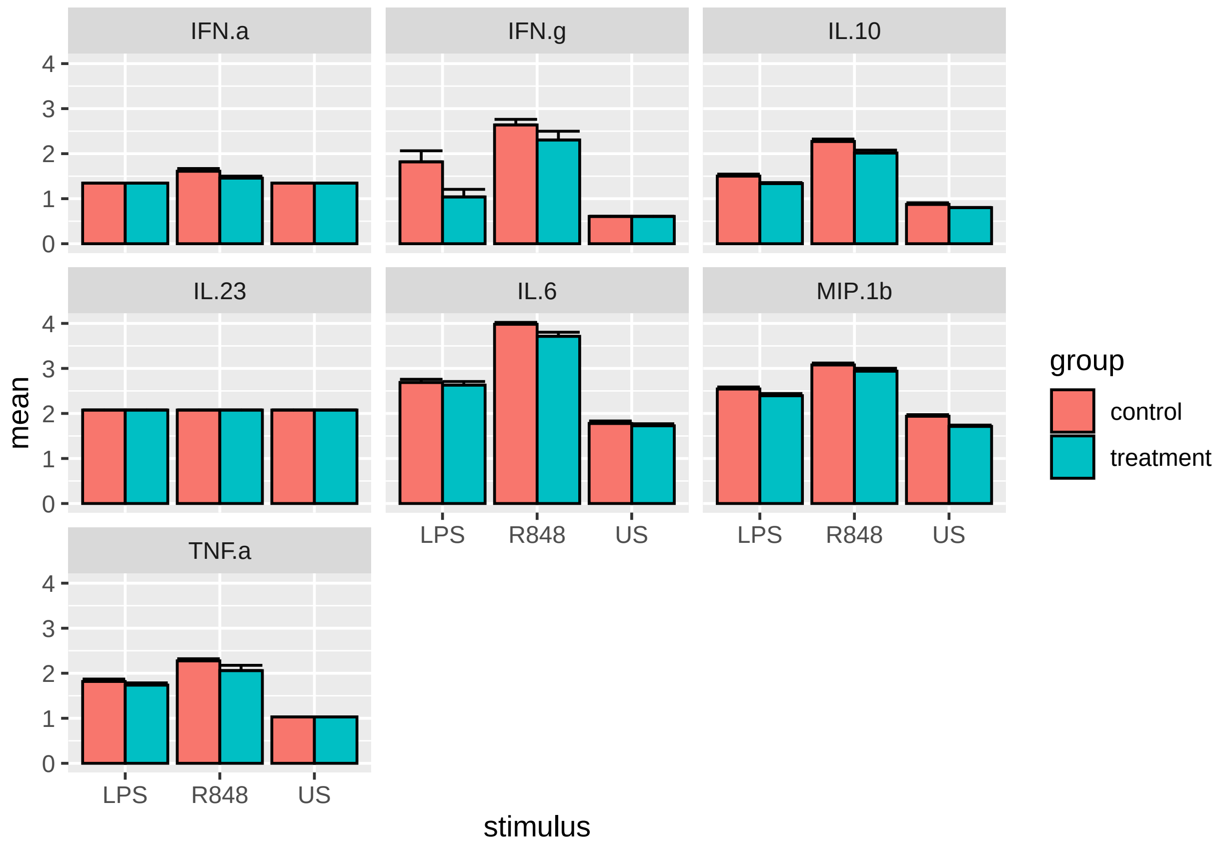
```
summarise(dat.meta,
          average = mean(`pg/mL`),
          standard_error = standard_error(`pg/mL`))
```

```
##    average standard_error
## 1 527.6194        96.6776
```

Here, summarise on it's own performed this for all the data at once. What ddply helped us do is subset the data by cytokine, group, and stimulus, summmarise the data, and return the results as one data frame (remember - ddply takes a data frame, performs a function, and returns another data frame). Building on the previous lesson, we can now used this 'summarized' dat.sum dataset to generate some very useful plots!

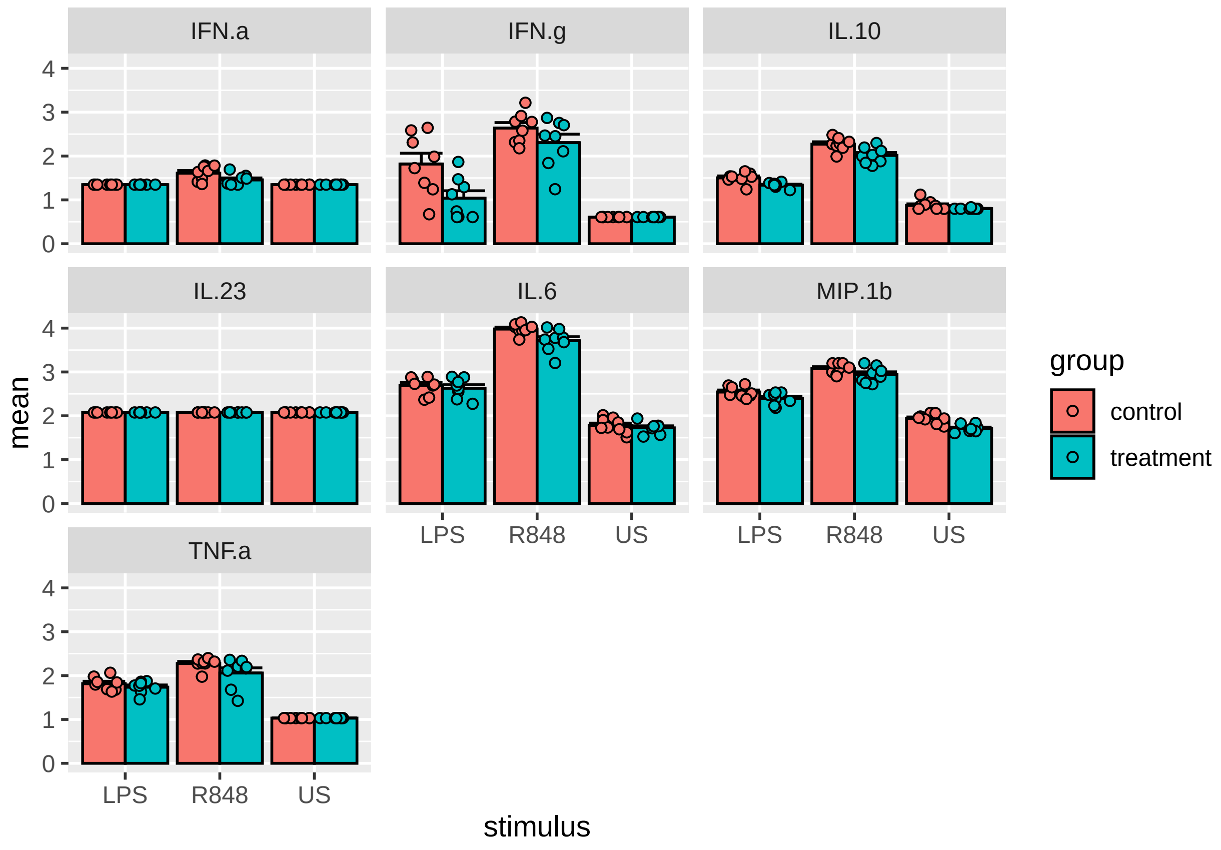
```
p <- ggplot(dat.sum, aes(x = stimulus, y = mean, fill  = group)) +
  geom_bar(color = "black", stat = "identity", position = "dodge") +
  geom_errorbar(aes(ymin = mean, ymax = mean + st.err), position = "dodge") +
  facet_wrap(~Cytokine)

p
```



The reason that we had to go through these extra steps is because ggplot prints bar graphs that take one of two statistics: count (think histogram), or identity (meaning the exact value in the cell - here, the mean). We have added the errorbars manually by specifying the upper bounds as the mean plus the standard error. However, ggplot is versatile. We can still add our data points back to the plot by using geom_point, and specifying our original, non-summarized data set.

```
p + geom_point(data = dat.meta,
               aes(x = stimulus, y = log10(1+`pg/mL`), fill = group),
               shape = 21,
               position = position_jitterdodge(dodge.width = 0.8, jitter.width = 0.3))
```



And that's it for today! Mission accomplished, and a few more skills placed under your belt in the process. You now know how to melt and cast, join, summarize, and plot! All skills that you will regularly employ.

```
##
##  -------------
## Go forth and melt!
##  --------------
##              \
##               \
##                \
##
##                  .="=.
##                _/.-.-.\_     _
##               ( ( o o ) )    ))
##                |/  "  \|    //
##                 \'---'/    //
##           jgs   /`"""`\\  ((
##                / /_,_\ \\  \\
##                \_\_'__/  \  ))
##                /`  /`~\   |//
##               /   /    \  /
##          ,--`,--'\/\    /
##          '-- "--'  '--'
```