Session 1: Data Assembly

Samuel P Callisto

June 7, 2018

Contents

Using R Markdown

Useful resource: R Markdown Cheatsheet https://www.rstudio.com/resources/cheatsheets/

Output in LaTeX

Surrounding \theta_{ij} with "$" will generate θ_{ij}

Useful keyboard shortcuts

(Mac Users use command instead of Ctrl)

Ctrl-Alt-I: Insert new chunk

Ctrl-Enter: Run current line of code (multiple if highlighted)

Ctrl-Shift-C: Comment/uncomment selected lines

Importing data

```
## load packages
library(dataTools)
library(tidyverse)

## Excel files
excel <- read.csv("datasets/TPM_sim_dataset_20180607.csv", as.is = T, stringsAsFactors = T, header = T)
```

Can also use file.choose() or absolute filepath (instead of relative to working directory) to select files you want to import.

Arguments to modify output

include, warning, message, echo, etc.

I always import ALL datasets and packages in my first chunk of code. This way you don't need to hunt down the line where data was imported, just re-run the first chunk.

Data types in R

There are multiple ways to represent data in R. Most of the time we work with numbers, and R uses different names for different types of numbers:

- integer or int: any whole number.
- numeric or num: numbers with a decimal. Also called double.

When we are working with text, there are multiple ways to represent them in R as well:

- character or chr: basic representation of a string
- Factor: R's representation of an enumerated data type (set of named levels)

Most of the time we want to work with text as a character because they are easier to manipulate. Numbers can also be treated as a character, but this is typically undesirable because we cannot perform any arithmetic manipulations on numbers when they are in this format.

```r
var <- as.integer(3)
str(var)
```

```
##  int 3
```

```r
var2 <- var*2
str(var2)
```

```
##  num 6
```

```r
var.c <- as.character(var)
str(var.c)
```

```
##  chr "3"
```

A third data type is the logical (or boolean) type (TRUE or T, FALSE or F). These are typically generated using logical tests.

Vectors contain the same data type. A vector of vectors create a matrix or a data.frame. Most of the time you will use data.frames to store your data since each vector can contain a different data type.

```r
vec.1 <- 1:5
str(vec.1)
```

```
##  int [1:5] 1 2 3 4 5
```

```r
vec.2 <- c("a", "b", "c", "d", "e")
str(vec.2)
```

```
##  chr [1:5] "a" "b" "c" "d" "e"
```

```r
vec.3 <- c(TRUE,FALSE,F,F,T)
str(vec.3)
```

```
##  logi [1:5] TRUE FALSE FALSE FALSE TRUE
```

```r
df <- data.frame(vec.1, vec.2, vec.3)
names(df) <- c("numbers", "letters", "vowels")

str(df)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ numbers: int  1 2 3 4 5
##  $ letters: Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5
##  $ vowels : logi  TRUE FALSE FALSE FALSE TRUE
```

```r
## convert from Factor to character data type
df$letters <- as.character(df$letters)
str(df)
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ numbers: int  1 2 3 4 5
##  $ letters: chr  "a" "b" "c" "d" ...
##  $ vowels : logi  TRUE FALSE FALSE FALSE TRUE
```

Some comments about reproducibility & readability

The importance of readability

A bad example of readability

```
write.csv(headr("datasets/TPM_sim_dataset_20180607.csv") %>%
  select(subjectid) %>%
  left_join(headr("datasets/Baseline_sim_dataset_AllSubjects_20180607.csv"),
            by= "subjectid"), "datasets/output.csv", row.names = F)
```

Can anyone guess what this line of code is accomplishing?

A good example of readability

```
## import data
baselineData <- read.csv("datasets/Baseline_sim_dataset_AllSubjects_20180607.csv")
topiramateData <- read.csv("datasets/TPM_sim_dataset_20180607.csv")

## create completers subset from topiramateData
completers <- select(topiramateData, subjectid)

## filter to keep only completers
filteredBaseline <- left_join(completers, baselineData, by= "subjectid")
```

```
## Warning: Column `subjectid` joining factors with different levels, coercing
## to character vector
```

```
## save filteredBaseline
# write.csv(filteredBaseline, "datasets/Baseline_sim_dataset_CompletersOnly_20180607.csv", row.names = 
```

Tip: Use “camel case” to separate words within a variable name by capitalizing the first letter of a new word, e.g. baselineData. This applies not only to coding within R, but also maintaining a proper audit trail for all your files!

Naming convention: for version control, best method of including dates is to use yyyymmdd, as this will sort chronologically when sorted alphabetically

Always write your code as though it were being read by a human, not a computer!

In addition to liberal commenting throughout your code, do not underestimate the importance of creating descriptive variable names. If a future graduate student takes over your project after you’ve graduated, will they be able to decipher what your code does without calling you to ask?

Version control

What's that mean?

Everytime you modify a file, you have created a new "version" of it. Version control allows you to keep a record of all the changes that you have made to a file. This way if you break a script that you have written, you can easily go back to the old version.

GitHub glossary

Commit: File changes saved to your local machine

Push: File changes saved to your online database

Pull: Bring online changes to your local workspace

Fork: Create a personal copy of someone else's work

Branch: Workspace to try implementing new feature without worrying about messing up existing function

Repositories

Can be either public or private. On github.com with a free account you can ONLY create public repositories. These can be seen by anyone. Enterprise or paid version of GitHub allows you to create private repositories which can only be seen by you and collaborators of your choice.

UMN GitHub

Students have free access to an Enterprise version of GitHub through UMN. Allows users to have private repositories. Users outside UMN cannot see anything in this account, but you can transfer repositories from the UMN GitHub to a public GitHub account.

Session 2: Data Manipulation

Dave Margraf

June 20, 2018

Contents

The `dplyr` package

The provides a grammar for data manipulation.

Load the dplyr package and data

```
library(dplyr)
```

Load the `Theoph` data set and save it as a data frame.

```
df <- data.frame(Theoph)
```

We can use the `dim()` and `head()` functions from base `R` to find the dimensions and take a look at the data.

```
dim(df)
```

```
## [1] 132   5
```

```
head(df)
```

```
##   Subject   Wt Dose Time  conc
## 1       1 79.6 4.02 0.00  0.74
## 2       1 79.6 4.02 0.25  2.84
## 3       1 79.6 4.02 0.57  6.57
## 4       1 79.6 4.02 1.12 10.50
## 5       1 79.6 4.02 2.02  9.66
## 6       1 79.6 4.02 3.82  8.58
```

Alternately, we can load the data as a tibble, which is a specialized data frame, with the `as_tibble()` function.

```
df <- as_tibble(Theoph)

df
```

```
## # A tibble: 132 x 5
##    Subject    Wt  Dose   Time  conc
##  * <ord>   <dbl> <dbl>  <dbl> <dbl>
##  1 1        79.6  4.02  0      0.74
##  2 1        79.6  4.02  0.25   2.84
##  3 1        79.6  4.02  0.570  6.57
##  4 1        79.6  4.02  1.12  10.5
##  5 1        79.6  4.02  2.02   9.66
##  6 1        79.6  4.02  3.82   8.58
##  7 1        79.6  4.02  5.1    8.36
##  8 1        79.6  4.02  7.03   7.47
##  9 1        79.6  4.02  9.05   6.89
## 10 1        79.6  4.02 12.1    5.94
## # ... with 122 more rows
```

Variable definitions for the `Theoph` data set:

- Wt - weight of the subject (kg)
- Dose - dose of theophylline administered orally to the subject (mg/kg)
- Time - time since drug administration when the sample was drawn (hr)
- conc - theophylline concentration in the sample (mg/L)

`filter()`

The filter verb subsets the data by rows (observations). That is, it extracts particular observations based their values.

Let's subset the theophylline data by weight of 70 kg or more.

```
filter(df, Wt >= 70)
```

```
## # A tibble: 77 x 5
##    Subject    Wt  Dose   Time  conc
##    <ord>   <dbl> <dbl>  <dbl> <dbl>
##  1 1        79.6  4.02  0      0.74
##  2 1        79.6  4.02  0.25   2.84
##  3 1        79.6  4.02  0.570  6.57
##  4 1        79.6  4.02  1.12  10.5
##  5 1        79.6  4.02  2.02   9.66
##  6 1        79.6  4.02  3.82   8.58
##  7 1        79.6  4.02  5.1    8.36
##  8 1        79.6  4.02  7.03   7.47
```

```
##   9 1          79.6   4.02   9.05    6.89
##  10 1          79.6   4.02  12.1     5.94
## # ... with 67 more rows
```

We can subset the data further with additional arguments.

```r
filter(df, Wt >= 70, Dose >= 4)
```

```
## # A tibble: 66 x 5
##     Subject     Wt  Dose   Time   conc
##     <ord>    <dbl> <dbl>  <dbl> <dbl>
##   1 1         79.6  4.02  0       0.74
##   2 1         79.6  4.02  0.25    2.84
##   3 1         79.6  4.02  0.570   6.57
##   4 1         79.6  4.02  1.12   10.5
##   5 1         79.6  4.02  2.02    9.66
##   6 1         79.6  4.02  3.82    8.58
##   7 1         79.6  4.02  5.1     8.36
##   8 1         79.6  4.02  7.03    7.47
##   9 1         79.6  4.02  9.05    6.89
##  10 1         79.6  4.02 12.1     5.94
## # ... with 56 more rows
```

`select()`

The select verb subsets the data by columns (variables). That is, it extracts particular variables based on their names.

We can extract a vector by naming one variable.

```r
select(df, conc)
```

```
## # A tibble: 132 x 1
##      conc
##   * <dbl>
##   1  0.74
##   2  2.84
##   3  6.57
##   4 10.5
##   5  9.66
##   6  8.58
##   7  8.36
##   8  7.47
##   9  6.89
##  10  5.94
## # ... with 122 more rows
```

We can drop variables as well. Just place a minus sign in front of the variable you want to remove. The other variables will remain.

```r
select(df, -Wt)
```

```
## # A tibble: 132 x 4
##     Subject  Dose   Time  conc
##   * <ord>   <dbl>  <dbl> <dbl>
##   1 1        4.02  0      0.74
##   2 1        4.02  0.25   2.84
```

```
##   3 1         4.02  0.570  6.57
##   4 1         4.02  1.12  10.5
##   5 1         4.02  2.02   9.66
##   6 1         4.02  3.82   8.58
##   7 1         4.02  5.1    8.36
##   8 1         4.02  7.03   7.47
##   9 1         4.02  9.05   6.89
##  10 1         4.02 12.1    5.94
## # ... with 122 more rows
```

Variables can be moved around if needed. Placing the `everything()` helper function will fill in the remaining variables you do not mention.

```
select(df, Time, Subject, everything())
```

```
## # A tibble: 132 x 5
##      Time Subject    Wt  Dose  conc
##   * <dbl> <ord>   <dbl> <dbl> <dbl>
##   1  0     1        79.6  4.02  0.74
##   2  0.25  1        79.6  4.02  2.84
##   3  0.570 1        79.6  4.02  6.57
##   4  1.12  1        79.6  4.02 10.5
##   5  2.02  1        79.6  4.02  9.66
##   6  3.82  1        79.6  4.02  8.58
##   7  5.1   1        79.6  4.02  8.36
##   8  7.03  1        79.6  4.02  7.47
##   9  9.05  1        79.6  4.02  6.89
##  10 12.1   1        79.6  4.02  5.94
## # ... with 122 more rows
```

If you want to move a variable to the end of the data set, subtract then add it back. Also, you can rename variables within any `select()` function.

```
select(df, -Wt, weight=Wt)
```

```
## # A tibble: 132 x 5
##    Subject  Dose   Time  conc weight
##   * <ord>   <dbl>  <dbl> <dbl>  <dbl>
##   1 1         4.02  0      0.74   79.6
##   2 1         4.02  0.25   2.84   79.6
##   3 1         4.02  0.570  6.57   79.6
##   4 1         4.02  1.12  10.5    79.6
##   5 1         4.02  2.02   9.66   79.6
##   6 1         4.02  3.82   8.58   79.6
##   7 1         4.02  5.1    8.36   79.6
##   8 1         4.02  7.03   7.47   79.6
##   9 1         4.02  9.05   6.89   79.6
##  10 1         4.02 12.1    5.94   79.6
## # ... with 122 more rows
```

rename()

The rename verb keeps all variables unlike select, which keeps only the variables you mention.

```
rename(df, weight = Wt)
```

```
## # A tibble: 132 x 5
##    Subject weight  Dose   Time  conc
##  * <ord>    <dbl> <dbl>  <dbl> <dbl>
##  1 1         79.6  4.02  0      0.74
##  2 1         79.6  4.02  0.25   2.84
##  3 1         79.6  4.02  0.570  6.57
##  4 1         79.6  4.02  1.12  10.5
##  5 1         79.6  4.02  2.02   9.66
##  6 1         79.6  4.02  3.82   8.58
##  7 1         79.6  4.02  5.1    8.36
##  8 1         79.6  4.02  7.03   7.47
##  9 1         79.6  4.02  9.05   6.89
## 10 1         79.6  4.02 12.1    5.94
## # ... with 122 more rows
```

An aside, the pipe operator %>%

Takes the result from the left hand side and passes it into the function on the right hand side. This allows you to code in a more readable left-to-right fashion rather than nesting function within one another. For example,

Let's practice using the using the filter verb to find the observations for the first subject.

```r
df %>% filter(Subject == 3)
```

```
## # A tibble: 11 x 5
##    Subject    Wt  Dose   Time  conc
##    <ord>   <dbl> <dbl>  <dbl> <dbl>
##  1 3        70.5  4.53  0      0
##  2 3        70.5  4.53  0.27   4.4
##  3 3        70.5  4.53  0.580  6.9
##  4 3        70.5  4.53  1.02   8.2
##  5 3        70.5  4.53  2.02   7.8
##  6 3        70.5  4.53  3.62   7.5
##  7 3        70.5  4.53  5.08   6.2
##  8 3        70.5  4.53  7.07   5.3
##  9 3        70.5  4.53  9      4.9
## 10 3        70.5  4.53 12.2    3.7
## 11 3        70.5  4.53 24.2    1.05
```

We can chain pipes together to really benefit from its usefulness. Find the observed Cmax for subject three.

```r
df %>%
  filter(Subject == 3) %>%
  select(conc) %>%
  max()
```

```
## [1] 8.2
```

`mutate()`

The mutate verb adds new variables.

New variables can be made that are functions of existing variables. For example, perhaps we want to express time in seconds rather than hours, or convert weight in kilograms to pounds.

Let's save this to `df` with the assignment operator `<-`.

```r
df <- df %>%
  mutate(minutes = Time * 60,
         lbs = Wt * 2.2046)

df
```

```
## # A tibble: 132 x 7
##    Subject    Wt  Dose   Time  conc minutes   lbs
##    <ord>   <dbl> <dbl>  <dbl> <dbl>   <dbl> <dbl>
##  1 1        79.6  4.02  0      0.74     0    175.
##  2 1        79.6  4.02  0.25   2.84    15    175.
##  3 1        79.6  4.02  0.570  6.57    34.2  175.
##  4 1        79.6  4.02  1.12  10.5     67.2  175.
##  5 1        79.6  4.02  2.02   9.66   121.   175.
##  6 1        79.6  4.02  3.82   8.58   229.   175.
##  7 1        79.6  4.02  5.1    8.36   306    175.
##  8 1        79.6  4.02  7.03   7.47   422.   175.
##  9 1        79.6  4.02  9.05   6.89   543    175.
## 10 1        79.6  4.02 12.1    5.94   727.   175.
## # ... with 122 more rows
```

arrange()

The arrange verb changes the ordering of the rows.

Sort the data by increasing weight.

```r
df %>% arrange(lbs)
```

```
## # A tibble: 132 x 7
##    Subject    Wt  Dose  Time  conc minutes   lbs
##    <ord>   <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl>
##  1 5        54.6  5.86  0     0        0    120.
##  2 5        54.6  5.86  0.3   2.02    18    120.
##  3 5        54.6  5.86  0.52  5.63    31.2  120.
##  4 5        54.6  5.86  1    11.4     60    120.
##  5 5        54.6  5.86  2.02  9.33   121.   120.
##  6 5        54.6  5.86  3.5   8.74   210    120.
##  7 5        54.6  5.86  5.02  7.56   301.   120.
##  8 5        54.6  5.86  7.02  7.09   421.   120.
##  9 5        54.6  5.86  9.1   5.9    546    120.
## 10 5        54.6  5.86 12     4.37   720    120.
## # ... with 122 more rows
```

Use `desc()` to sort a variable in descending order.

```r
df %>% arrange(desc(lbs))
```

```
## # A tibble: 132 x 7
##    Subject    Wt  Dose  Time  conc minutes   lbs
##    <ord>   <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl>
##  1 9        86.4   3.1  0     0        0    190.
##  2 9        86.4   3.1  0.3   7.37    18    190.
##  3 9        86.4   3.1  0.63  9.03    37.8  190.
##  4 9        86.4   3.1  1.05  7.14    63    190.
```

```
##   5 9          86.4   3.1   2.02  6.33   121.   190.
##   6 9          86.4   3.1   3.53  5.66   212.   190.
##   7 9          86.4   3.1   5.02  5.67   301.   190.
##   8 9          86.4   3.1   7.17  4.24   430.   190.
##   9 9          86.4   3.1   8.8   4.11   528    190.
##  10 9          86.4   3.1  11.6   3.16   696    190.
## # ... with 122 more rows
```

Adding verbs together.

```
df %>%
  filter(Time ==0) %>%
  arrange(desc(lbs))
```

```
## # A tibble: 12 x 7
##    Subject    Wt  Dose  Time  conc minutes   lbs
##    <ord>   <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl>
##   1 9        86.4  3.1      0  0          0  190.
##   2 6        80    4        0  0          0  176.
##   3 1        79.6  4.02     0  0.74       0  175.
##   4 4        72.7  4.4      0  0          0  160.
##   5 2        72.4  4.4      0  0          0  160.
##   6 3        70.5  4.53     0  0          0  155.
##   7 8        70.5  4.53     0  0          0  155.
##   8 11       65    4.92     0  0          0  143.
##   9 7        64.6  4.95     0  0.15       0  142.
##  10 12       60.5  5.3      0  0          0  133.
##  11 10       58.2  5.5      0  0.24       0  128.
##  12 5        54.6  5.86     0  0          0  120.
```

By subsetting and sorting the data we can see that three subjects have positive drug concentrations at time zero, and dose appears to be inversely proportional to weight.

group_by()

You will usually want to group data by some variable.

Grouping doesn't change how the data looks (apart from listing how it's grouped), but it does change how it acts with the other dplyr verbs.

```
df %>%
  group_by(Subject)
```

```
## # A tibble: 132 x 7
## # Groups:   Subject [12]
##    Subject    Wt  Dose  Time   conc minutes   lbs
##    <ord>   <dbl> <dbl> <dbl>  <dbl>   <dbl> <dbl>
##   1 1        79.6  4.02  0      0.74     0     175.
##   2 1        79.6  4.02  0.25   2.84    15     175.
##   3 1        79.6  4.02  0.570  6.57    34.2   175.
##   4 1        79.6  4.02  1.12  10.5     67.2   175.
##   5 1        79.6  4.02  2.02   9.66   121.    175.
##   6 1        79.6  4.02  3.82   8.58   229.    175.
##   7 1        79.6  4.02  5.1    8.36   306     175.
##   8 1        79.6  4.02  7.03   7.47   422.    175.
##   9 1        79.6  4.02  9.05   6.89   543     175.
```

```
## 10 1         79.6  4.02 12.1    5.94   727.   175.
## # ... with 122 more rows
```

Now we can create a new columns specific to each subject with `group_by()` and `mutate()`. Let's find the Cmax and Tmax for each concentration-time profile. Since Tmax is related to the pharmacokinetic parameter Cmax, we can use the `case_when()` function to identify the time when Cmax is observed. This observation is saved in a temporary variable, `temp`, then dropped with the `select()` verb.

```
df %>%
  group_by(Subject) %>%
  mutate(Cmax = max(conc),
         temp = case_when(conc == Cmax ~ Time),
         Tmax = max(temp, na.rm = T)) %>%
  select(-temp)
```

```
## # A tibble: 132 x 9
## # Groups:   Subject [12]
##    Subject    Wt  Dose   Time  conc minutes   lbs  Cmax  Tmax
##    <ord>   <dbl> <dbl>  <dbl> <dbl>   <dbl> <dbl> <dbl> <dbl>
##  1 1        79.6  4.02  0      0.74     0    175.  10.5  1.12
##  2 1        79.6  4.02  0.25   2.84    15    175.  10.5  1.12
##  3 1        79.6  4.02  0.570  6.57    34.2  175.  10.5  1.12
##  4 1        79.6  4.02  1.12  10.5     67.2  175.  10.5  1.12
##  5 1        79.6  4.02  2.02   9.66   121.   175.  10.5  1.12
##  6 1        79.6  4.02  3.82   8.58   229.   175.  10.5  1.12
##  7 1        79.6  4.02  5.1    8.36   306    175.  10.5  1.12
##  8 1        79.6  4.02  7.03   7.47   422.   175.  10.5  1.12
##  9 1        79.6  4.02  9.05   6.89   543    175.  10.5  1.12
## 10 1        79.6  4.02 12.1    5.94   727.   175.  10.5  1.12
## # ... with 122 more rows
```

`summarise()`

The summarise verb reduces multiple values down to a single summary.

```
df %>%
  summarise(meanWt = mean(Wt),
            medWt = median(Wt),
            n = n_distinct(Subject))
```

```
## # A tibble: 1 x 3
##   meanWt medWt     n
##    <dbl> <dbl> <int>
## 1   69.6  70.5    12
```

You may want to group data before summarizing.

```
df %>%
  group_by(Wt < 70) %>%
  summarise(medDose = median(Dose),
            meanDose = mean(Dose),
            sdDose = sd(Dose))
```

```
## # A tibble: 2 x 4
##   `Wt < 70` medDose meanDose sdDose
##   <lgl>       <dbl>    <dbl>  <dbl>
```

```
## 1 FALSE         4.4     4.14  0.474
## 2 TRUE          5.3     5.31  0.355
```

Let's build a data set

Some useful base `R` functions:

`seq()` generates regular sequences.
`rep()` replicates values.
`length()` gets or sets the length of vectors (including lists) and factors.
`unique()` returns a vector, data frame or array like `x` but with duplicate elements/rows removed.
`sample()` takes a random sample of the specified size from the elements of `x` either with or without replacement.
`round()` rounds the values to the specified number of decimal places (default 0).

Statistical functions in the `stats` package.

`rnorm()` random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`. `runif()` generates random deviates about the uniform distribution on the interval from `min` to `max`.

Subjects

To create a vector for 20 subjects we can start with the `seq()` function.

```r
seq(1:20)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

If we want longitudinal (repeated measures) data we can pipe this into the `rep()` function.

```r
seq(1:20) %>% rep(10)
```

```
##   [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20  1  2  3
##  [24]  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20  1  2  3  4  5  6
##  [47]  7  8  9 10 11 12 13 14 15 16 17 18 19 20  1  2  3  4  5  6  7  8  9
##  [70] 10 11 12 13 14 15 16 17 18 19 20  1  2  3  4  5  6  7  8  9 10 11 12
##  [93] 13 14 15 16 17 18 19 20  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
## [116] 16 17 18 19 20  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
## [139] 19 20  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20  1
## [162]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20  1  2  3  4
## [185]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

This isn't quite right. We could use `arrange()` to fix this, but an easier way is to use the `each =` argument in `rep()`. Note: using `rep(10)` is equivalent to `rep(times=10)`.

```r
  seq(1:20) %>% rep(each=10)
```

```
##   [1]  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2  3  3  3
##  [24]  3  3  3  3  3  3  3  4  4  4  4  4  4  4  4  4  4  5  5  5  5  5  5
##  [47]  5  5  5  5  6  6  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7
##  [70]  7  8  8  8  8  8  8  8  8  8  8  9  9  9  9  9  9  9  9  9  9 10 10
##  [93] 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11 11 11 12 12 12 12 12
## [116] 12 12 12 12 12 13 13 13 13 13 13 13 13 13 13 14 14 14 14 14 14 14 14
## [139] 14 14 15 15 15 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16 16 16 17
```

```
## [162] 17 17 17 17 17 17 17 17 17 18 18 18 18 18 18 18 18 18 18 19 19 19 19
## [185] 19 19 19 19 19 19 20 20 20 20 20 20 20 20 20 20
```

That looks better. Let's store this in a tibble named `new`.

```
new <- seq(1:20) %>%
        rep(each=10) %>%
        as_tibble()
```

Sampling times

Next, we'll create a vector of sampling times.

```
c(0,1,2,3,4,6,9,12,18,24)
```

```
##  [1]  0  1  2  3  4  6  9 12 18 24
```

Use the `rep()` function to match `id` and save it as the variable `time`.

```
time <-
  c(0,1,2,3,4,6,9,12,18,24) %>%
  rep(20)
```

We can add this to the data set with `mutate()` and change the name of `value` to `id` with `rename()`.

```
new <- new %>%
  rename(id = value) %>%
  mutate(time = time)

new
```

```
## # A tibble: 200 x 2
##       id  time
##    <int> <dbl>
##  1     1     0
##  2     1     1
##  3     1     2
##  4     1     3
##  5     1     4
##  6     1     6
##  7     1     9
##  8     1    12
##  9     1    18
## 10     1    24
## # ... with 190 more rows
```

This is a good start but how often are sampling times this precise? We can add some variability and create a new variable. Sample from the the normal distribution with a mean of 1 and a small standard deviation, multiply by nominal time, then round the result.

```
timeR <- time %>%
  '*'(rnorm(200,1,0.05)) %>%
  round(2)

new <- new %>%
      rename(nomTime = time) %>%
      mutate(time = timeR)
```

```r
new
```

```
## # A tibble: 200 x 3
##       id nomTime  time
##    <int>   <dbl> <dbl>
##  1     1       0  0
##  2     1       1  1.05
##  3     1       2  1.93
##  4     1       3  2.91
##  5     1       4  4.31
##  6     1       6  6.1
##  7     1       9  8.77
##  8     1      12 12.3
##  9     1      18 19.0
## 10     1      24 26
## # ... with 190 more rows
```

Simulating binary or categorical variables with equal probability of being chosen

```r
sample(c(0,1), length(unique(new$id)), replace = T) %>% rep(each=10)
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
##  [36] 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
##  [71] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [141] 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0
## [176] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```r
sample(c(1,2,3,4), length(unique(new$id)), replace = T) %>% rep(each=10)
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
##  [36] 3 3 3 3 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3
##  [71] 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4
## [106] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4
## [141] 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 3 3 3 3 3
## [176] 3 3 3 3 3 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
```

Use `set.seed()` for reproducible results.

```r
set.seed(1907)
sex <- sample(c(0,1), length(unique(new$id)), replace = T) %>% rep(each=10)
```

```r
set.seed(1907)
race <- sample(c(1,2,3,4), length(unique(new$id)), replace = T) %>% rep(each=10)

new <- new %>%
    mutate(sex = sex,
           race)

new
```

```
## # A tibble: 200 x 5
```

```
##        id nomTime  time   sex  race
##     <int>   <dbl> <dbl> <dbl> <dbl>
##   1     1       0  0         0     1
##   2     1       1  1.05      0     1
##   3     1       2  1.93      0     1
##   4     1       3  2.91      0     1
##   5     1       4  4.31      0     1
##   6     1       6  6.1       0     1
##   7     1       9  8.77      0     1
##   8     1      12 12.3       0     1
##   9     1      18 19.0       0     1
## 10     1      24 26         0     1
## # ... with 190 more rows
```

Note the argument in 'mutate()' to keep the same variable name.

Simulate a uniform distribution of ages

```
set.seed(1907)
age <- runif(length(unique(new$id)), 18, 65) %>% rep(each=10) %>% floor()

age
```

```
##   [1] 18 18 18 18 18 18 18 18 18 18 55 55 55 55 55 55 55 55 55 55 26 26 26
##  [24] 26 26 26 26 26 26 26 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45 45
##  [47] 45 45 45 45 43 43 43 43 43 43 43 43 43 43 45 45 45 45 45 45 45 45 45
##  [70] 45 26 26 26 26 26 26 26 26 26 26 41 41 41 41 41 41 41 41 41 41 44 44
##  [93] 44 44 44 44 44 44 44 44 20 20 20 20 20 20 20 20 20 20 28 28 28 28 28
## [116] 28 28 28 28 28 37 37 37 37 37 37 37 37 37 37 51 51 51 51 51 51 51 51
## [139] 51 51 45 45 45 45 45 45 45 45 45 45 29 29 29 29 29 29 29 29 29 29 18
## [162] 18 18 18 18 18 18 18 18 18 22 22 22 22 22 22 22 22 22 22 43 43 43 43
## [185] 43 43 43 43 43 43 54 54 54 54 54 54 54 54 54 54
```

```
new <- new %>% mutate(age)

new
```

```
## # A tibble: 200 x 6
##        id nomTime  time   sex  race   age
##     <int>   <dbl> <dbl> <dbl> <dbl> <dbl>
##   1     1       0  0         0     1    18
##   2     1       1  1.05      0     1    18
##   3     1       2  1.93      0     1    18
##   4     1       3  2.91      0     1    18
##   5     1       4  4.31      0     1    18
##   6     1       6  6.1       0     1    18
##   7     1       9  8.77      0     1    18
##   8     1      12 12.3       0     1    18
##   9     1      18 19.0       0     1    18
## 10     1      24 26         0     1    18
## # ... with 190 more rows
```

Check the documentation for `round()` to look at the `floor()` function and others related to it.

Finding first and last observations for a subject in longitudial data

```
new <- new %>%
  mutate(fid = as.numeric(!duplicated(new$id)),
         lid = as.numeric(!duplicated(new$id, fromLast = T)))

new
```

```
## # A tibble: 200 x 8
##       id nomTime  time   sex  race   age   fid   lid
##    <int>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1     1       0  0        0     1    18     1     0
##  2     1       1  1.05     0     1    18     0     0
##  3     1       2  1.93     0     1    18     0     0
##  4     1       3  2.91     0     1    18     0     0
##  5     1       4  4.31     0     1    18     0     0
##  6     1       6  6.1      0     1    18     0     0
##  7     1       9  8.77     0     1    18     0     0
##  8     1      12 12.3      0     1    18     0     0
##  9     1      18 19.0      0     1    18     0     0
## 10     1      24 26        0     1    18     0     1
## # ... with 190 more rows
```

Exercise: Summarize the new dataset.

Session information

```
sessionInfo()
```

```
## R version 3.5.1 (2018-07-02)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 17134)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] bindrcpp_0.2.2 dplyr_0.7.6
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.17     knitr_1.20       bindr_0.1.1      magrittr_1.5
##  [5] tidyselect_0.2.4 R6_2.2.2         rlang_0.2.1      stringr_1.3.1
##  [9] tools_3.5.1      utf8_1.1.4       cli_1.0.0        htmltools_0.3.6
```

```
## [13] yaml_2.2.0       assertthat_0.2.0 rprojroot_1.3-2  digest_0.6.15
## [17] tibble_1.4.2     crayon_1.3.4     purrr_0.2.5      glue_1.2.0
## [21] evaluate_0.10.1  rmarkdown_1.10   stringi_1.2.3    compiler_3.5.1
## [25] pillar_1.2.3     backports_1.1.2  pkgconfig_2.0.1
```

Session 3: A consize guide to ggplot2

Ashwin Karanam

June 27, 2018

Contents

1 The `ggplot2` package

A graphing package in the tidyverse based on "The Grammar of Graphics" (Lelanad Wilkson). It uses a layered structure to graphing which simplifies the process of coding plots in R. Intuitively, we know that any graph looks like Figure 1. ggplot adopts this concept.

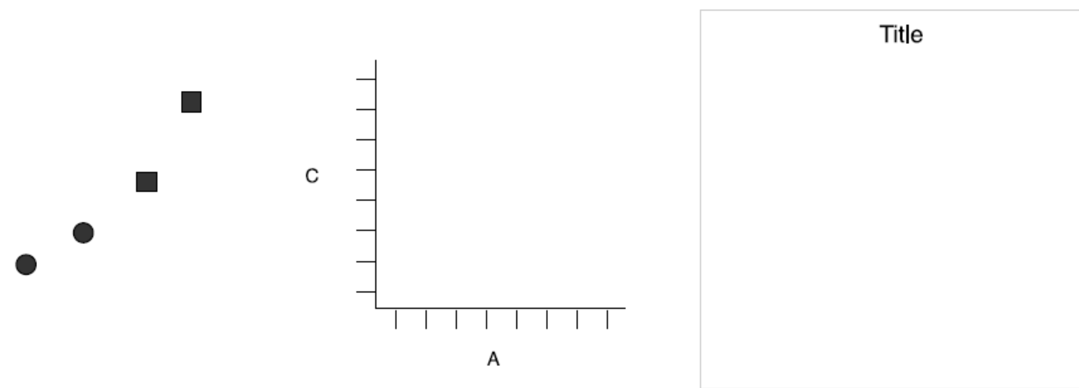


Figure 1. Graphics objects produced by (from left to right): geometric objects, scales and coordinate system, plot annotations.

1.1 The Grammar in ggplot2

Essential layers that dictate a plot:

1. a default dataset and set of mappings from variables to aesthetics,
2. one or more layers, with each layer having one geometric object, one statistical transformation,
3. one position adjustment, and optionally, one dataset and set of aesthetic mappings,
4. one scale for each aesthetic mapping used,
5. a coordinate system,
6. the facet specification

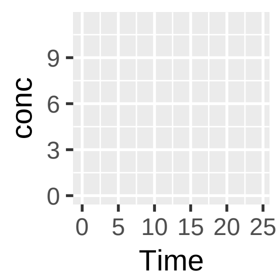
2 Hands on with ggplot2

Because we are interested in R for PMx, lets use PK data. The thoephylline data is good enough.

```
library(ggplot2)
library(dplyr)
library(gridExtra)
library(mrgsolve)

df <- data.frame(Theoph)
```

As we have already worked with this data before, let's skip the introductions and get right into the coding. Use the `ggplot()` function to create the first layer which is the base plot.

```
ggplot(df,
       aes(Time,conc))
```

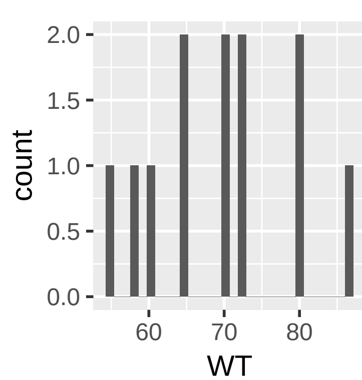


Every plot should start with `ggplot()` as this maps the x and y axis along with which data to use. To this we add layers that inform what type of plot to create using "geoms".

2.1 Histograms

Let's try a basic histogram plot of subject weights. We will use the `geom_histogram()` for this.

```
wt <- df %>%
  group_by(df$Subject) %>%
  summarise(WT = mean(Wt))

ggplot(wt,aes(WT))+
  geom_histogram()
```



There are several options within `geom_histogram()` that let you modify the histograms.

2.2 Scatterplots/Trendplots

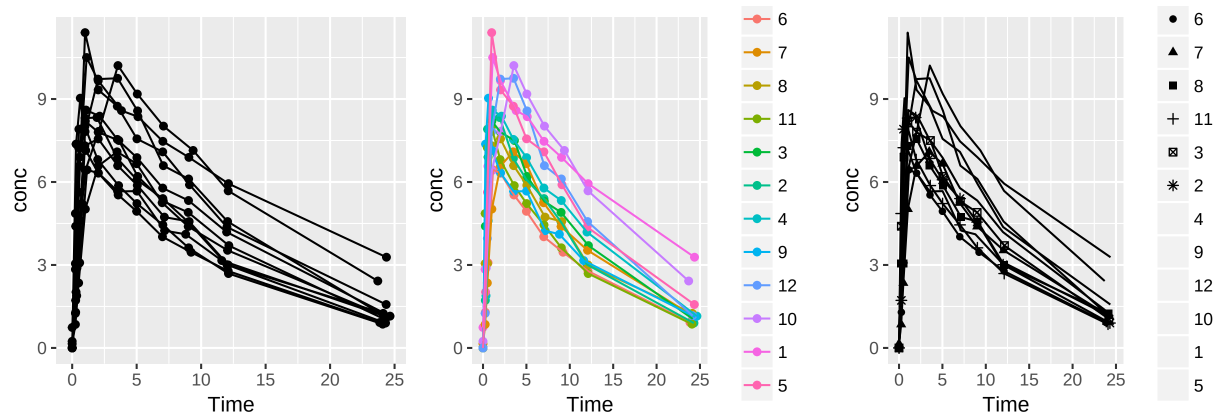
2.2.1 Sphagetti plots

Trendplots are one of the most important type of plots in PMx. `geom_line` is used for connecting observations in the order they appear in. The "group" argument allows you to map a group of factors. The other way of doing this would be to use the color and linetype arguments.

```
a <- ggplot(df,aes(Time,conc,group=Subject))+
  geom_point()+
  geom_line()

b <- ggplot(df,aes(Time,conc,color=Subject))+
  geom_point()+
  geom_line()

c <- ggplot(df,aes(Time,conc,group=Subject))+
  geom_point(aes(shape=Subject))+
  geom_line()

grid.arrange(a,b,c,ncol=3)
```

```
## Warning: The shape palette can deal with a maximum of 6 discrete values
## because more than 6 becomes difficult to discriminate; you have
## 12. Consider specifying shapes manually if you must have them.
```

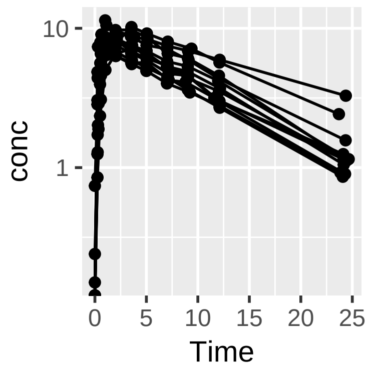
```
## Warning: Removed 66 rows containing missing values (geom_point).
```



As you see they do the same thing as group, but assigns a different color/linetype to each group. Of course this is not such a good idea because of limited number of shapes/lines!

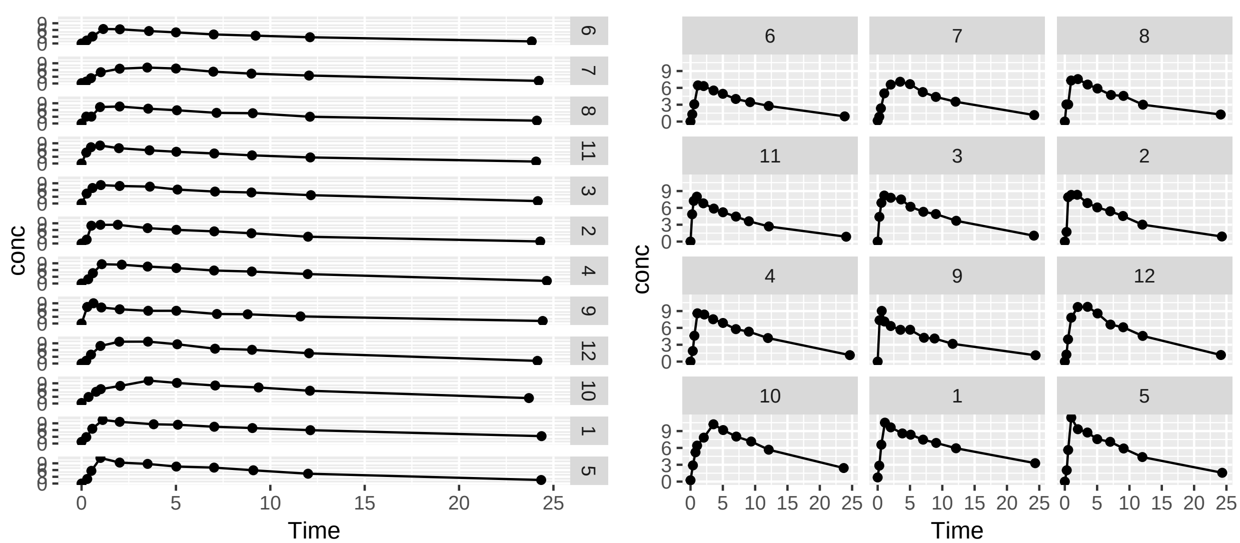
Another commonly used transformation is for the y-axis. For log scale Y-axis, we use the `scale_y_log10()` function which is an offshoot of `scale_y_continuous()`. Similar functions are available for x-axis transformations.

```
ggplot(df,aes(Time,conc,group=Subject))+
  geom_point()+
  geom_line()+
  scale_y_log10()
```



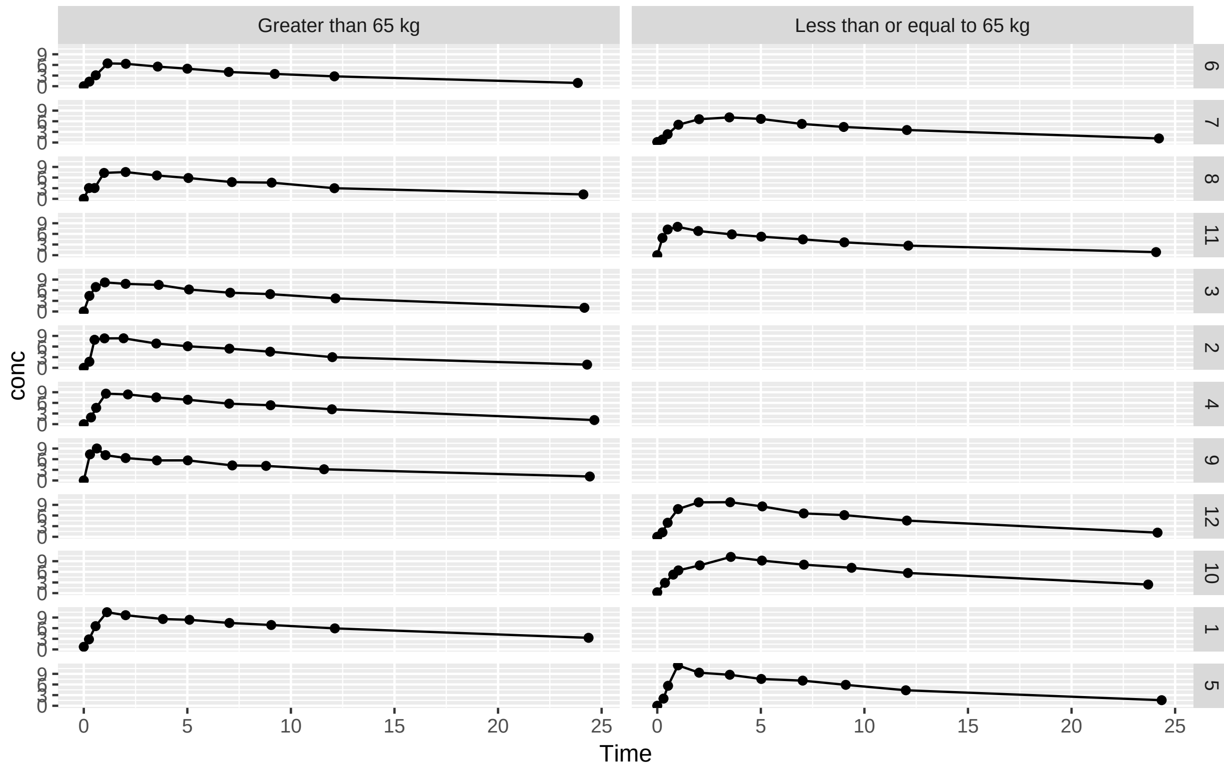
Now, let's create one plot for each individual. `facet_grid()` is one way to facet your data. This is a good verb to use for small IDs, but becomes useless when use large datasets. The other verb useful here is `facet_wrap()` which allows you to customize your grid.

```
c <- ggplot(df,aes(Time,conc))+
  geom_point()+
  geom_line()+
  facet_grid(Subject~.)

d <- ggplot(df,aes(Time,conc))+
  geom_point()+
  geom_line()+
  facet_wrap(~Subject,ncol=3)

grid.arrange(c,d,ncol=2)
```



We can add a second layer of faceting to this.

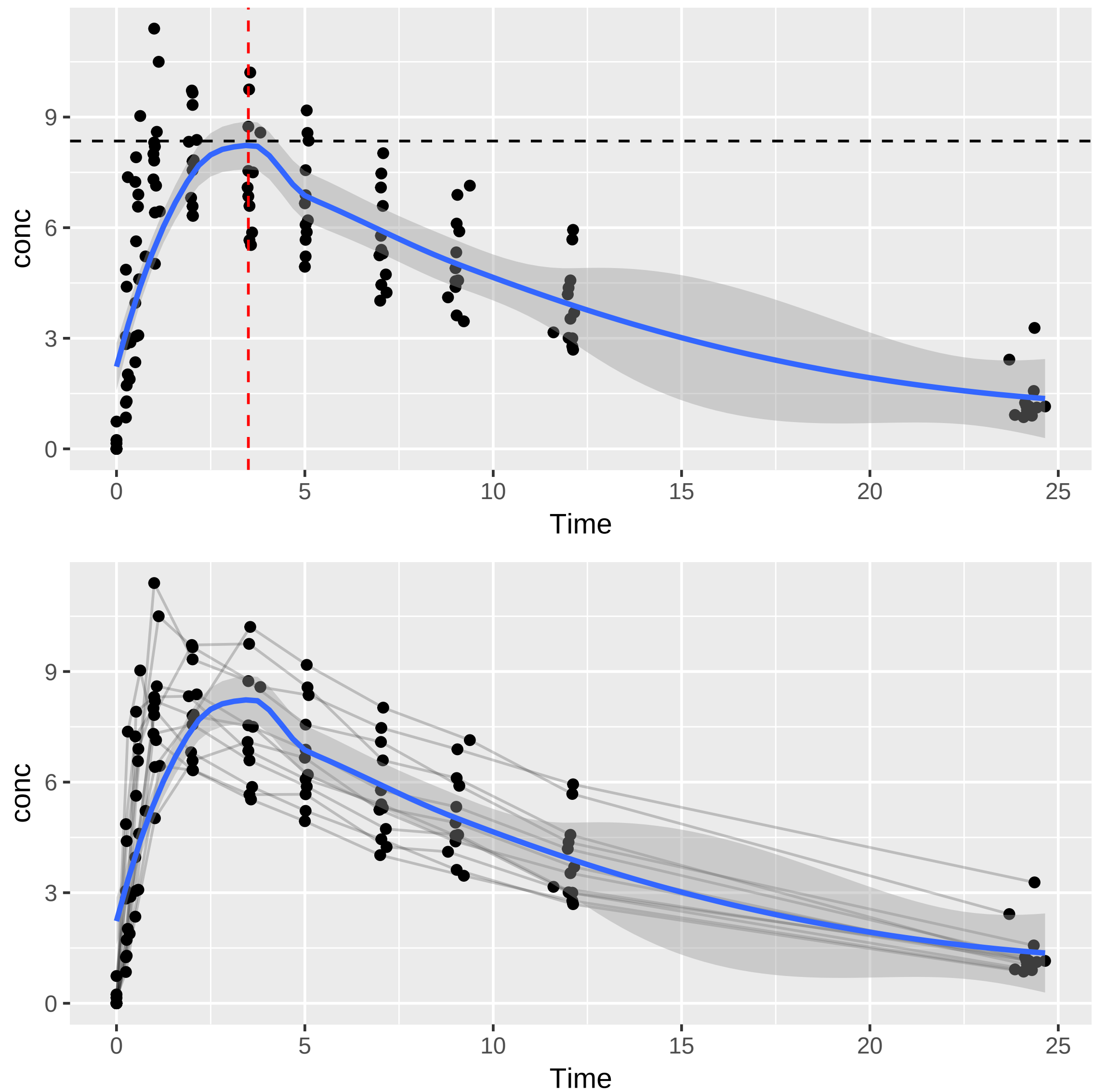
```
df2 <- df %>%
  mutate(Category = if_else(Wt <= 65,
                            "Less than or equal to 65 kg",
                            "Greater than 65 kg"))

ggplot(df2,aes(Time,conc))+
  geom_point()+
  geom_line()+
  facet_grid(Subject~Category)
```



2.2.2 Trendplots

With the same concentration time plot, you can add a trend line eg. a non-parametric smoother. The `geom_smooth` verb allows for this in ggplot.

```
e <- ggplot(df,aes(Time,conc))+
  geom_point()+
  geom_smooth(se=TRUE,method="loess")+
  geom_vline(xintercept = 3.5, linetype=2,color="red")+
  geom_hline(yintercept = 8.35, linetype=2,color="black")

f <- ggplot(df,aes(Time,conc))+
  geom_point()+
  geom_line(aes(group=Subject),alpha=0.2)+
  geom_smooth(se=TRUE,method="loess")

grid.arrange(e,f,ncol=1)
```



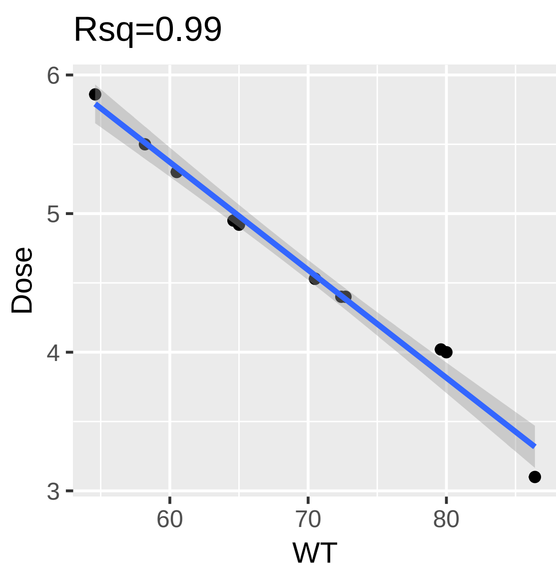
Couple of things to notice in the first plot:

1. the grouping asthetic is shifted to geom_point as geom_smooth does not need the grouping asthetic.
2. By default geom_smooth plots the SE and uses the loess method. Additional methods include lm, glm and gam.
3. Use `geom_hline()` and `geom_vline()` to create reference lines.

The second plot here is adding individual sphagetti plots to the average plot. I am using the `aplha` argument in the geom_line to make the individual lines lighter. You can use this for any geom available.

While we are on smoothers, let me introduce you to correlation plots. Let's say we want a correlation plot for weight and dose.

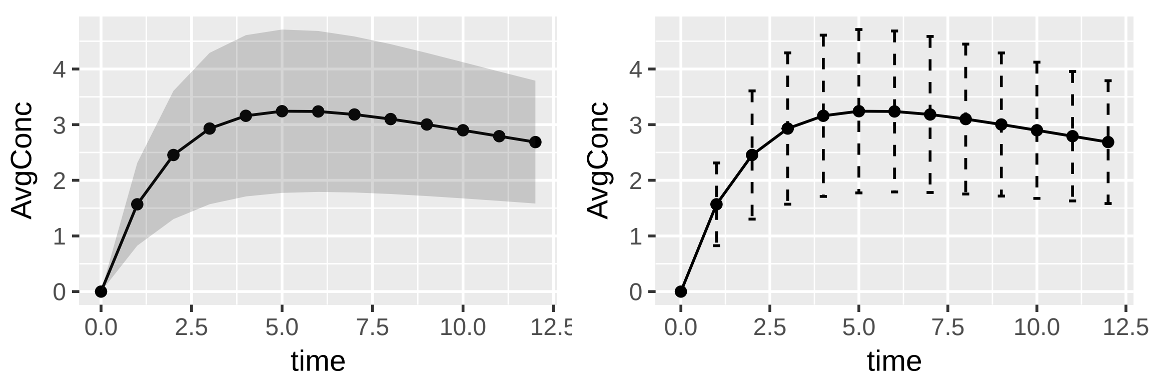
```
wt <- df %>%
  dplyr::group_by(Subject) %>%
  dplyr::summarise(WT = mean(Wt),Dose=mean(Dose))

cor <- cor(wt$WT,wt$Dose)

ggplot(wt,aes(WT,Dose))+
  geom_point()+
  geom_smooth(method="lm")+
  labs(title="Rsq=0.99")
```



Do not use these in-lieu of actual statistical analysis. Confirm your linear fits by running regression analysis.

Another common type of plot is an average trend plot with standard deviation. Use `geom_ribbon()` for creating a shaded region in your plot. We can also use `geom_errorbar()` for this which would give you error bars around the mean. Let us use something from mrgsolve for this exercise.

```
mod <- mread_cache("popExample",modlib())

idata <- data.frame(ID=1:10)
mat1 <- dmat(0.1,0.2)

set.seed(12358)
out <- mod %>%
  omat(mat1) %>%
  idata_set(object="idata")%>%
  ev(amt=100)%>%
  obsonly()%>%
  mrgsim(end=12,delta=1) %>%
  as.data.frame()

avg <- out %>%
  dplyr::group_by(time)%>%
  dplyr::summarise(AvgConc = mean(DV),
                   SD=sd(DV))

g <- ggplot(avg,aes(time,AvgConc))+
  geom_point()+
  geom_line()+
  geom_ribbon(aes(ymin=AvgConc-SD,ymax=AvgConc+SD),alpha=0.2)

h <- ggplot(avg,aes(time,AvgConc))+
  geom_point()+
  geom_line()+
  geom_errorbar(aes(ymin=AvgConc-SD,ymax=AvgConc+SD),
                linetype=2,
                size=0.5,
                width=0.2)

grid.arrange(g,h,ncol=2)
```

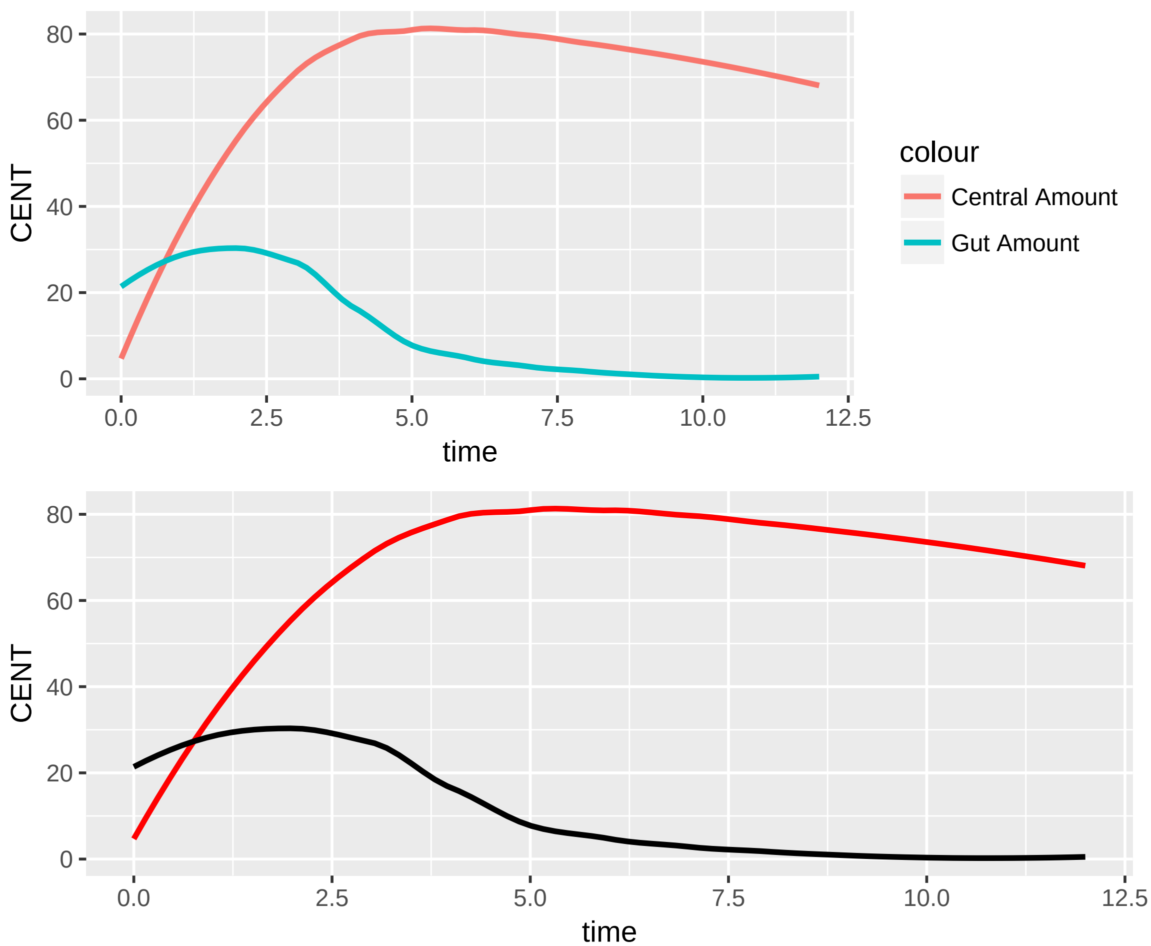


2.3 Adding multiple trend lines

2.3.1 Multiple smoothers of the same type of data

Use `geom_smooth` for adding multiple trends. Just add a seperate `geom_smooth` with new mappings.

```
i <- ggplot(out, aes(time,CENT))+
  geom_smooth(aes(color="Central Amount"), se=FALSE)+
  geom_smooth(aes(time, GUT,color="Gut Amount"), se=FALSE)

j <- ggplot(out, aes(time,CENT))+
  geom_smooth(color="red", se=FALSE)+
  geom_smooth(aes(time, GUT),color="black", se=FALSE)


grid.arrange(i,j,ncol=1)
```

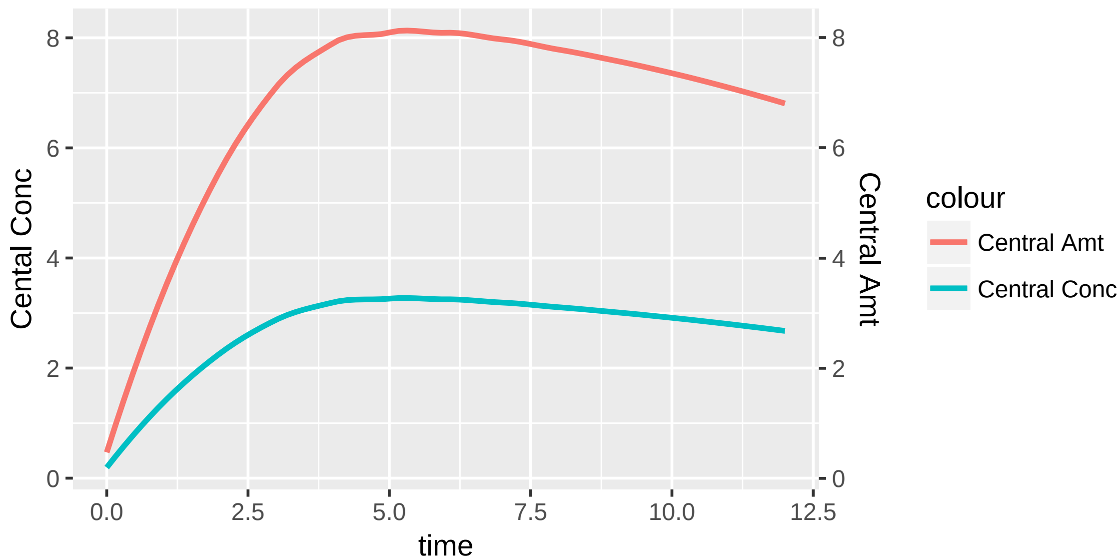


If you notice, in the first plot the color argument is within the aesthetics argument. This forces ggplot to consider all mappings in that geom as one entity. Here we are forcing ggplot to call the first layer as central concentration and second as central amount. In the second plot, we do not use the aesthetics mapping for color but use the RGB manual color selection. This does not give us a legend because you manually specified which layer has which color.

2.3.2 Multiple smoothers of different types of data

If you want to simulatenously plot a concentration and an amount curve, you'll need two Y-axis. `sec.axis` is the verb to create a secondary axis.

```
ggplot(out, aes(time,DV))+
  geom_smooth(aes(color="Central Conc"), se=FALSE)+
  geom_smooth(aes(time,(CENT/10),color="Central Amt"),se=FALSE)+
  scale_y_continuous( name="Cental Conc",
                      sec.axis = sec_axis(trans = ~. *1, name = "Central Amt"))
```



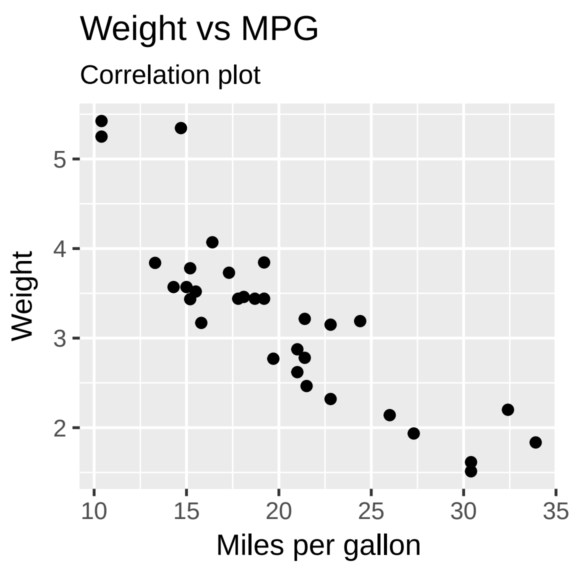
You can also use `sec_axis()` to create a secondary axis, be it for x or y axes.

2.4 Formatting in ggplot

2.4.1 Formatting the axis labels

Use the `labs()` to edit the plot title, and axis labels. You can also use `xlab()`, `ylab()` and `ggtitle()` to individually edit these. For this exercise let us use the "mtcars" dataset

```
cars <- data.frame(mtcars)

ggplot(cars, aes(mpg, wt))+
  geom_point()+
  labs(title="Weight vs MPG",
       subtitle="Correlation plot",
       x="Miles per gallon",
       y="Weight")
```

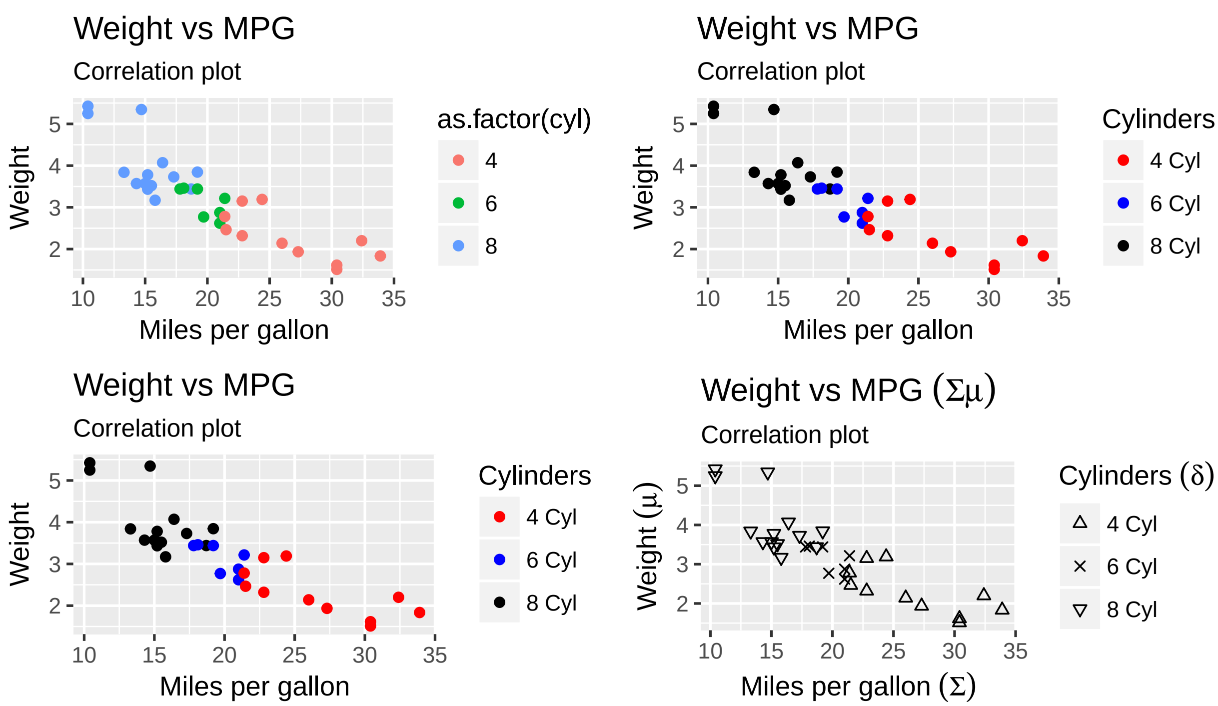


2.4.2 Formatting legends

ggplot uses the names of your columns or factors in said colums to determine the names of the legends. An esay way to override those is to use the `scale_color_manual()` verb. This allows you to choose customize not only the names and titles, but also the color. Similar functions are available for shape `scale_shape_manual()`, fill `scale_fill_manual()`, size `scale_size_manual()`, linetype `scale_linetype_manual()` and alpha `scale_alpha_manual()`. Similar off-shoots are available for discrete variables `scale_*_discrete()` and continious variables `scale_*_continuous()`.

```
k <- ggplot(cars, aes(mpg, wt, color=as.factor(cyl)))+
  geom_point()+
  labs(title="Weight vs MPG",
       subtitle="Correlation plot",
       x="Miles per gallon",
       y="Weight")

l <- ggplot(cars, aes(mpg, wt, color=as.factor(cyl)))+
  geom_point()+
  labs(title="Weight vs MPG",
       subtitle="Correlation plot",
       x="Miles per gallon",
       y="Weight")+
  scale_colour_manual(values=c("red", "blue", "black"),
                      name="Cylinders",
                      breaks=c("4", "6", "8"),
                      labels=c("4 Cyl", "6 Cyl", "8 Cyl"))

m <- ggplot(cars, aes(mpg, wt, color=as.factor(cyl)))+
  geom_point()+
  labs(title="Weight vs MPG",
       subtitle="Correlation plot",
       x="Miles per gallon",
       y="Weight")+
  scale_colour_manual(values=c("red", "blue", "black"),
```

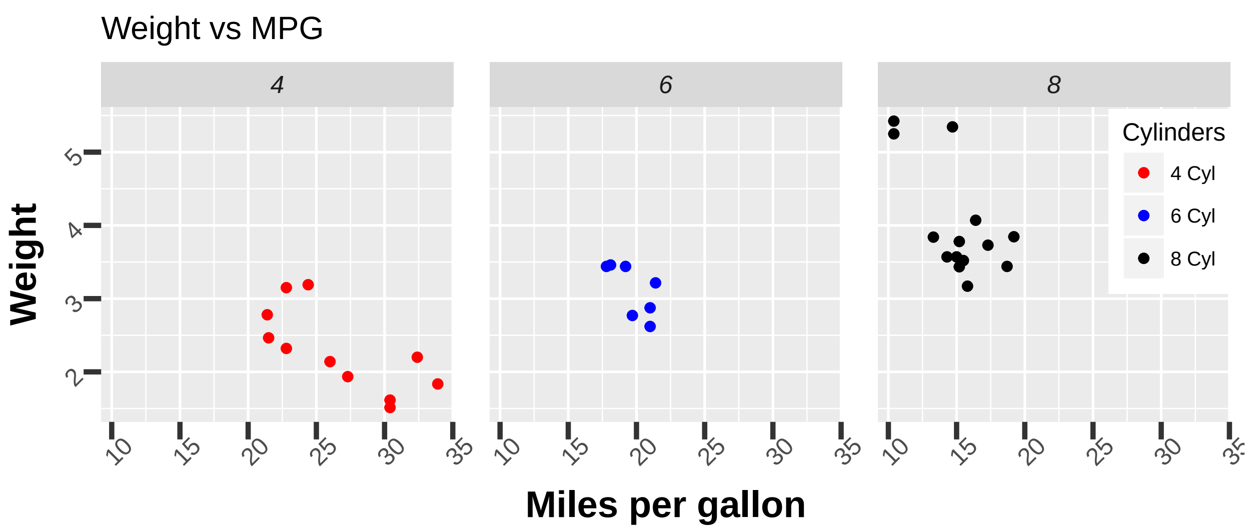
```
                            name="Cylinders",
                            breaks=c("4", "6", "8"),
                            labels=c("4 Cyl", "6 Cyl", "8 Cyl"))

n <- ggplot(cars, aes(mpg, wt, shape=as.factor(cyl)))+
  geom_point()+
  labs(title="Weight vs MPG"~(Sigma*mu),
       subtitle="Correlation plot",
       x="Miles per gallon"~(Sigma),
       y="Weight"~(mu))+
  scale_shape_manual(values=c(2,4,6),
                            name="Cylinders"~(delta),
                            breaks=c("4", "6", "8"),
                            labels=c("4 Cyl","6 Cyl","8 Cyl"))

grid.arrange(k,l,m,n,ncol=2)
```



2.4.3 Text size formatting

`theme()` is a great way of formatting all your text in one place. There are around 50 arguments within `theme()` all aimed at providing user control over the formatting.

```
ggplot(cars, aes(mpg, wt, color=as.factor(cyl)))+
  geom_point()+
  facet_grid(.~as.factor(cyl))+
  labs(title="Weight vs MPG",
       x="Miles per gallon",
       y="Weight")+
  scale_colour_manual(values=c("red", "blue", "black"),
                      name="Cylinders",
                      breaks=c("4", "6", "8"),
                      labels=c("4 Cyl", "6 Cyl", "8 Cyl")) +
  theme(axis.title.x=element_text(size=15,face="bold"),
        axis.title.y=element_text(size=15,face="bold"),
        axis.text = element_text(size=10,angle = 45),
        axis.ticks = element_line(size = 1),
        axis.ticks.length = unit(.25, "cm"),
        strip.text.x = element_text(size=10,face="italic"),
        legend.position = c(0.95,0.7),
        legend.title=element_text(size=10),
        legend.text=element_text(size=8),
        panel.spacing = unit(1, "lines"))
```



3 Colors, themes and exporting with ggplot2

3.1 R Color

This is a short section on colors and how to custmomize colors in R and in ggplot2

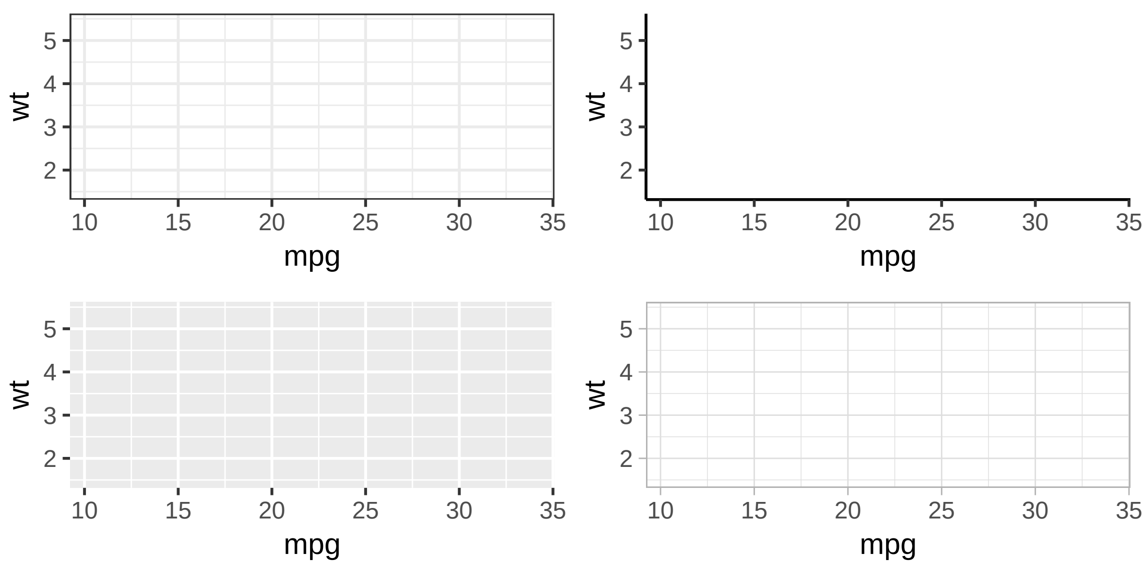
There are several help or how-to sections on the web about R colors. There are countless times when the defaults in ggplot look decent for a presentation, but when you print it out, they look really bad. Overcome them by:

1. Here is a consize guide to the color choices are given by `Columbia-Stats`. These colors go hand in hand with `scale_color_manual()`
2. If you're a fan of using the color pallete instead of the names, then I suggest use this `link`
3. If you want to be adventurous, there's several packages that give fun palletes like `wesanderson`

3.2 Themes in ggplot

There are several built in themes availabe for ggplot. Use the `theme_*()` verb to explore the options. Here are some for demostration.

```
o <- ggplot(cars, aes(mpg, wt, shape=as.factor(cyl)))+
  theme_bw()

p <- ggplot(cars, aes(mpg, wt, shape=as.factor(cyl)))+
  theme_classic()

q <- ggplot(cars, aes(mpg, wt, shape=as.factor(cyl)))+
  theme_grey()

r <- ggplot(cars, aes(mpg, wt, shape=as.factor(cyl)))+
  theme_light()

grid.arrange(o,p,q,r,ncol=2)
```



3.3 Exporting publication quality images

You can output almost any type of image file. The verbs are `tiff()`, `bmp()`, `jpeg()`, and `png()`.

```
tiff("test.tiff", width = 9, height = 8, units = 'in', res = 300)
l
dev.off()
```

4 Helpful References

1. `R in Action` by Robert Kabacoff
2. `Hands-On Programming` with R by Garrett Grolemund
3. `R Cookbook` by Paul Teetor
4. `THE ART OF R PROGRAMMING` by Norman Matloff
5. `The Grammar of Graphics` by Leland Wilkinson
6. `A Layered Grammar of Graphics` by Hadley Wickham

5 Session Information

```
sessionInfo()
```

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 17134)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=English_United States.1252
## [2] LC_CTYPE=English_United States.1252
## [3] LC_MONETARY=English_United States.1252
## [4] LC_NUMERIC=C
## [5] LC_TIME=English_United States.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] bindrcpp_0.2         mrgsolve_0.8.10.9007 gridExtra_2.3
## [4] dplyr_0.7.4          ggplot2_2.2.1
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.15              bindr_0.1
##  [3] knitr_1.20                magrittr_1.5
##  [5] munsell_0.4.3             colorspace_1.3-2
##  [7] R6_2.2.2                  RcppArmadillo_0.8.300.1.0
##  [9] rlang_0.2.0               stringr_1.3.1
## [11] plyr_1.8.4                tools_3.4.3
## [13] grid_3.4.3                gtable_0.2.0
## [15] htmltools_0.3.6           yaml_2.1.16
## [17] lazyeval_0.2.1            rprojroot_1.3-2
```

```
## [19] digest_0.6.15          assertthat_0.2.0
## [21] tibble_1.4.2           reshape2_1.4.3
## [23] glue_1.2.0             evaluate_0.10.1
## [25] rmarkdown_1.8          labeling_0.3
## [27] stringi_1.1.7          compiler_3.4.3
## [29] pillar_1.1.0           scales_0.5.0
## [31] backports_1.1.2        pkgconfig_2.0.1
```

Session 5: xpose

Ashwin Karanam

Contents

1 The xpose package

Xpose is an R-based model building aid for population analysis using NONMEM. It facilitates data set checkout, exploration and visualization, model diagnostics, candidate covariate identification and model comparison created by Andrew Hooker, Mats O. Karlsson, Benjamin Guiastrennec and E. Niclas Jonsson from Uppsala University.

1.1 Install package

```
# Install the lastest release from the CRAN
install.packages('xpose')

# Or install the development version from GitHub
# install.packages('devtools')
devtools::install_github('UUPharmacometrics/xpose')
```

1.2 xpose requirements

To make full use of the functionality offered by `xpose` the following NONMEM output files should be available:

- `.lst/.out/.res`: used to collect information on the run (`template_titles`) as well as the output table names. Alternatively a model file (`.mod/.ctl`) can be used but some of the information in `template_titles` may not be available.
- `.ext`: used to collect final parameter estimates and residual standard error (RSE)
- `.phi`: used for the random effects and iOFV
- `.cov`: used for the covariance matrix
- `.cor`: used for the correlation matrix
- `.grd`: used for the estimation gradients
- `.shk`: used to compute random effect shrinkage `template_titles`
- output and simulation tables: for the actual data

When importing the files, xpose will return messages to the console and inform of any issue encountered during the import.

`xpose` is compatible with the `$TABLE FIRSTONLY` option of NONMEM. The option `FIRSTONLY` only output the first record for each `ID` and hence can be used to decrease the size of output tables having no time-varying columns. During tables import xpose will merge `FIRSTONLY` tables with regular tables allowing seamless use of columns from `FIRSTONLY` in plots.

1.3 Create xpose database

```
library(xpose)
library(tidyverse)
library(gridExtra)

#xpdb_ex_pk is an inbuilt example from xpose.
# Look at the ~/Documents/R/win-library/3.4/xpose/extdata
xpdb <- xpdb_ex_pk
```

If your run number is `001`, all your NONMEM output files will end in `001`. Create xpose database using this:

```
xpdb <- xpose_data(runno = '001')

Looking for nonmem output tables
Reading: sdtab001, catab001, cotab001, patab001 [$prob no.1]

Looking for nonmem output files
Reading: run001.cor, run001.cov, run001.ext, run001.grd, run001.phi
```

These messages can be silenced with the option `quiet = TRUE`.

2 xpose xpdb data access

A typical `xpdb` object contains 8 levels namely:

- `code`: the parsed model code
- `summary`: contains key information regarding the model. All the information contained in the summary can be used as part of the `template_titles`.
- `data`: contains all output and simulation tables as well as the column indexing
- `files`: contains all output files
- `special`: contains post-processed datasets used by functions like `vpc()`
- `gg_theme`: an attached ggplot2 theme
- `xp_theme`: an attached xpose theme
- `options`: attached global options

2.1 Glimpse at the xpdb

The files attached to an xpdb object can be displayed to the console simply by writing the xpdb name to the console or by using the `print()` function. Any of these files can be accessed from the xpdb using one of the functions listed below.

```
xpdb # or print(xpdb)
```

```
## run001.lst overview:
##  - Software: nonmem 7.3.0
##  - Attached files (memory usage 1.3 Mb):
##    + obs tabs: $prob no.1: catab001.csv, cotab001, patab001, sdtab001
##    + sim tabs: $prob no.2: simtab001.zip
##    + output files: run001.cor, run001.cov, run001.ext, run001.grd, run001.phi, run001.shk
##    + special: <none>
##  - gg_theme: theme_readable
##  - xp_theme: theme_xp_default
##  - Options: dir = analysis/models/pk/, quiet = FALSE, manual_import = NULL
```

2.2 Access model code

The `get_code()` function can be used to access the parsed model code from the xpdb. This code was used to create the summary and find table names. The parsed code can be used to get additional information about the run. If the argument `.problem` is specified a subset of the code can be returned based on `$PROBLEM`.

Note that general code warnings and PsN outputs appended are listed as problem 0.

```r
code <- get_code(xpdb)
code
```

```
## # A tibble: 764 x 5
##    problem level subroutine code                         comment
##  *   <int> <int> <chr>      <chr>                        <chr>
##  1       0     0 oth        Mon Oct 16 13:34:28 CEST 20~ ""
##  2       0     0 oth        ""                           ; 1. Based on: 0~
##  3       0     0 oth        ""                           "; 2. Descriptio~
##  4       0     0 oth        ""                           ; NONMEM PK exam~
##  5       1     1 pro        Parameter estimation         ""
##  6       1     2 inp        ID DOSE DV SCR AGE SEX CLAS~ ""
##  7       1     2 inp        " CLCR AMT SS II EVID"       ""
##  8       1     3 dat        ../../mx19_2.csv IGNORE=@    ""
##  9       1     4 abb        DERIV2=NO                    ""
## 10       1     5 sub        ADVAN2 TRANS1                ""
## # ... with 754 more rows
```

2.3 Access the output data

The `get_data()` function can be used to access the imported table files. Tables can be accessed by `table` name or by `.problem`. In the latter a single dataset containing all aggregated tables is returned. If more than one `table` name or `.problem` number is provided a named list is returned.

Note when providing a table name it is not guaranteed that the table will be identical to its file (i.e. the order of the columns may have been changed and tables with ***FIRSTONLY*** *will no longer be deduplicated).*

```r
data <- get_data(xpdb, table = 'patab001')
data
```

```
## # A tibble: 550 x 8
##    ID       KA    CL     V ALAG1    ETA1     ETA2    ETA3
##    <fct> <dbl> <dbl> <dbl> <dbl>   <dbl>    <dbl>   <dbl>
##  1 110   0.496  25.5   141 0.208 -0.0370 -0.00596 -2.14
##  2 110   0.496  25.5   141 0.208 -0.0370 -0.00596 -2.14
##  3 110   0.496  25.5   141 0.208 -0.0370 -0.00596 -2.14
##  4 110   0.496  25.5   141 0.208 -0.0370 -0.00596 -2.14
##  5 110   0.496  25.5   141 0.208 -0.0370 -0.00596 -2.14
##  6 110   0.496  25.5   141 0.208 -0.0370 -0.00596 -2.14
##  7 110   0.496  25.5   141 0.208 -0.0370 -0.00596 -2.14
##  8 112   4.11   21.8   122 0.208 -0.0495  0.122   -0.0235
##  9 112   4.11   21.8   122 0.208 -0.0495  0.122   -0.0235
## 10 112   4.11   21.8   122 0.208 -0.0495  0.122   -0.0235
## # ... with 540 more rows
```

2.4 Access the run files

The `get_file()` function can be used to access the imported output files. Files can be accessed by `file` name, by `.problem`, `.subprob` and/or `.method`. If more than one `file` name, `.problem`, `.subprob`, or `.method` is provided a named list is returned.

```r
file <- get_file(xpdb, file = 'run001.ext')
file
```

```
## # A tibble: 28 x 16
##    ITERATION THETA1 THETA2 THETA3 THETA4 THETA5  THETA6  THETA7
##        <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>   <dbl>   <dbl>
##  1      0      25.4   1.47   7.45  0.214  0.200 0.00983 0.00601
##  2      1.00   26.3   1.26   7.35  0.219  0.217 0.00989 0.00602
##  3      2.00   25.6   1.47   7.29  0.216  0.212 0.00987 0.00603
##  4      3.00   26.8   1.49   5.76  0.213  0.213 0.00979 0.00628
##  5      4.00   26.7   1.49   5.69  0.213  0.212 0.00979 0.00629
##  6      5.00   26.7   1.49   5.66  0.213  0.212 0.00979 0.00630
##  7      6.00   26.6   1.49   5.03  0.210  0.217 0.0100  0.00652
##  8      7.00   26.6   1.49   4.93  0.205  0.217 0.0100  0.00658
##  9      8.00   26.6   1.48   4.62  0.211  0.217 0.00951 0.00735
## 10      9.00   26.6   1.46   4.41  0.209  0.217 0.00903 0.00874
## # ... with 18 more rows, and 8 more variables: `SIGMA(1,1)` <dbl>,
## #   `OMEGA(1,1)` <dbl>, `OMEGA(2,1)` <dbl>, `OMEGA(2,2)` <dbl>,
## #   `OMEGA(3,1)` <dbl>, `OMEGA(3,2)` <dbl>, `OMEGA(3,3)` <dbl>, OBJ <dbl>
```

2.5 Access the parameter estimates

The `get_prm()` function can be used to access the parameter estimates. To get a nice parameter table printed to the console use the function `prm_table()` instead. The arguments `.problem`, `.subprob` and `.method` can be used to select the parameter estimates to output.

```
# Raw output for editing
prm <- get_prm(xpdb, digits = 4)
prm
```

```
## # A tibble: 11 x 10
##    type  name   label    value      se     rse fixed diagonal     m     n
##  * <chr> <chr>  <chr>    <dbl>    <dbl>   <dbl> <lgl> <lgl>    <dbl> <dbl>
##  1 the   THETA1 TVCL   2.63e+1  0.892    0.0339 F     NA        1.00 NA
##  2 the   THETA2 TVV    1.35e+0  0.0438   0.0325 F     NA        2.00 NA
##  3 the   THETA3 TVKA   4.20e+0  0.809    0.192  F     NA        3.00 NA
##  4 the   THETA4 LAG    2.08e-1  0.0157   0.0755 F     NA        4.00 NA
##  5 the   THETA5 Prop.~ 2.05e-1  0.0224   0.110  F     NA        5.00 NA
##  6 the   THETA6 Add. ~ 1.06e-2  0.00366  0.347  F     NA        6.00 NA
##  7 the   THETA7 CRCL ~ 7.17e-3  0.00170  0.237  F     NA        7.00 NA
##  8 ome   OMEGA~ IIV CL 2.70e-1  0.0233   0.0862 F     T         1.00  1.00
##  9 ome   OMEGA~ IIV V  1.95e-1  0.0320   0.164  F     T         2.00  2.00
## 10 ome   OMEGA~ IIV KA 1.38e+0  0.202    0.146  F     T         3.00  3.00
## 11 sig   SIGMA~ ""     1.00e+0 NA       NA      T     T         1.00  1.00
```

```
# Nicely formated table
prm_table(xpdb, digits = 4)
```

```
##
## Reporting transformed parameters:
## For the OMEGA and SIGMA matrices, values are reported as standard deviations for the diagonal elemen
##
## Estimates for $prob no.1, subprob no.0, method foce
##  Parameter  Label      Value        RSE
##  THETA1     TVCL       26.29        0.03391
##  THETA2     TVV        1.348        0.0325
##  THETA3     TVKA       4.204        0.1925
##  THETA4     LAG        0.208        0.07554
```

```
##  THETA5     Prop. Err  0.2046       0.1097
##  THETA6     Add. Err   0.01055      0.3466
##  THETA7     CRCL on CL 0.007172     0.2366
##  OMEGA(1,1) IIV CL     0.2701       0.08616
##  OMEGA(2,2) IIV V      0.195        0.1643
##  OMEGA(3,3) IIV KA     1.381        0.1463
##  SIGMA(1,1)            1        fix  -
```

For the OMEGA and SIGMA matrices, values are reported as standard deviations for the diagonal elements and as correlations for the off-diagonal elements. The relative standard errors (RSE) for OMEGA and SIGMA are reported on the approximate standard deviation scale (SE/variance estimate)/2. Use `transform = FALSE` to report untransformed parameters.

2.6 Access the run summary

The `get_summary()` function can be used to access the generated run summary from which the `template_titles`. If the argument `.problem` is specified a subset of the summary can be returned based on `$PROBLEM`.

Note that general summary information are listed as problem 0.

```
run_sum <- get_summary(xpdb, .problem = 0)
run_sum
```

```
## # A tibble: 12 x 5
##    problem subprob descr                label       value
##      <dbl>   <dbl> <chr>                <chr>       <chr>
##  1       0       0 Run description      descr       NONMEM PK example for ~
##  2       0       0 Run directory        dir         analysis/models/pk/
##  3       0       0 Run errors           errors      na
##  4       0       0 ESAMPLE seed number  esampleseed na
##  5       0       0 Run file             file        run001.lst
##  6       0       0 Number of ESAMPLE    nesample    na
##  7       0       0 Reference model      ref         000
##  8       0       0 Run number           run         run001
##  9       0       0 Software             software    nonmem
## 10       0       0 Run start time       timestart   Mon Oct 16 13:34:28 CE~
## 11       0       0 Run stop time        timestop    Mon Oct 16 13:34:35 CE~
## 12       0       0 Software version     version     7.3.0
```

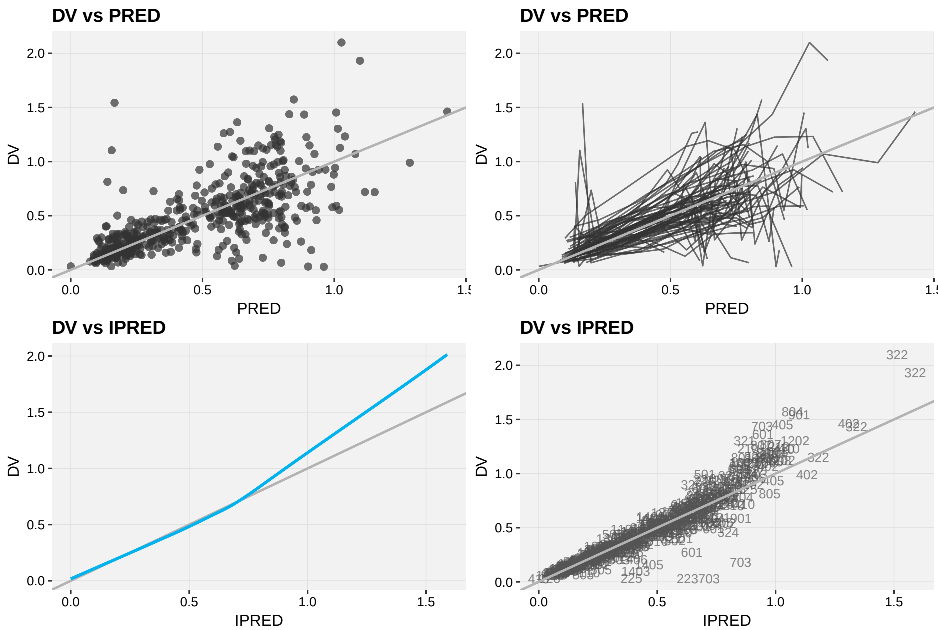
3 Basic GOF plots

3.1 Scatter plots

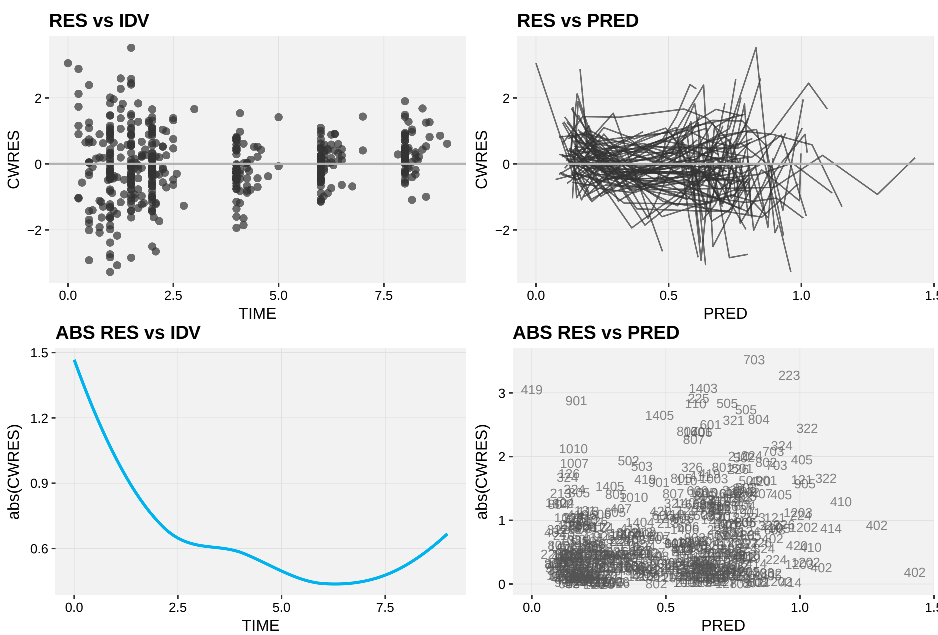
Use the `dv_vs_xxx()` to create basic scatter plots for `IPRED` and `PRED`. Use `type=` to switch between line `l`, point `p`, smooth `s` and text `t`. Similarly residual scatter plots can be created by using `res_vs_xxx()` or `absval_res_vs_xxx()` for `IDV` and `PRED`.

```
gridExtra::grid.arrange(
  dv_vs_pred(xpdb, title = "DV vs PRED", subtitle = NULL, caption = NULL, type = 'p'),
  dv_vs_pred(xpdb, title = "DV vs PRED", subtitle = NULL, caption = NULL, type = 'l'),
  dv_vs_ipred(xpdb, title = "DV vs IPRED", subtitle = NULL, caption = NULL, type = 's'),
  dv_vs_ipred(xpdb, title = "DV vs IPRED", subtitle = NULL, caption = NULL, type = 't'),
```

```
  ncol = 2
)
```



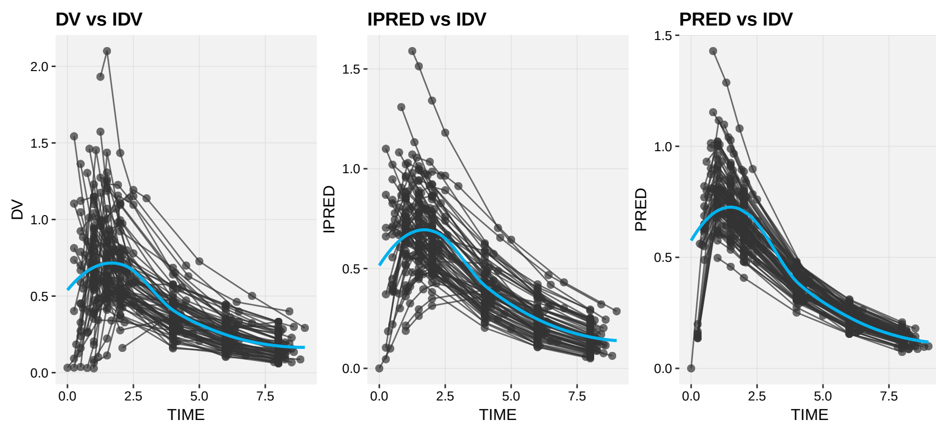

```
gridExtra::grid.arrange(
  res_vs_idv(xpdb, title = "RES vs IDV", subtitle = NULL, caption = NULL, type = 'p'),
  res_vs_pred(xpdb, title = "RES vs PRED", subtitle = NULL, caption = NULL, type = 'l'),
  absval_res_vs_idv(xpdb,title="ABS RES vs IDV",
                    subtitle = NULL,caption = NULL,type = 's'),
  absval_res_vs_pred(xpdb,title="ABS RES vs PRED",
                     subtitle = NULL,caption = NULL,type = 't'),
  ncol = 2
)
```



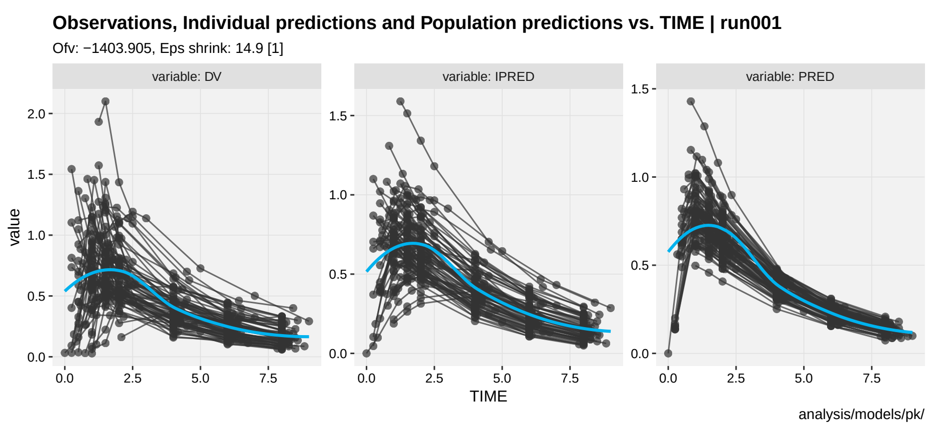
3.2 More scatter plots

Use the `xxx_vs_idv` to plot scatter with trends for `DV`, `IPRED` and `PRED`. The `dv_preds_vs_idv()` lets you plot DV, PRED and IPRED side by side.

```
gridExtra::grid.arrange(
  dv_vs_idv(xpdb, title = "DV vs IDV", subtitle = NULL, caption = NULL),
  ipred_vs_idv(xpdb, title = "IPRED vs IDV",
               subtitle = NULL, caption = NULL),
  pred_vs_idv(xpdb, title = "PRED vs IDV",
              subtitle = NULL, caption = NULL),
  ncol = 3
)
```



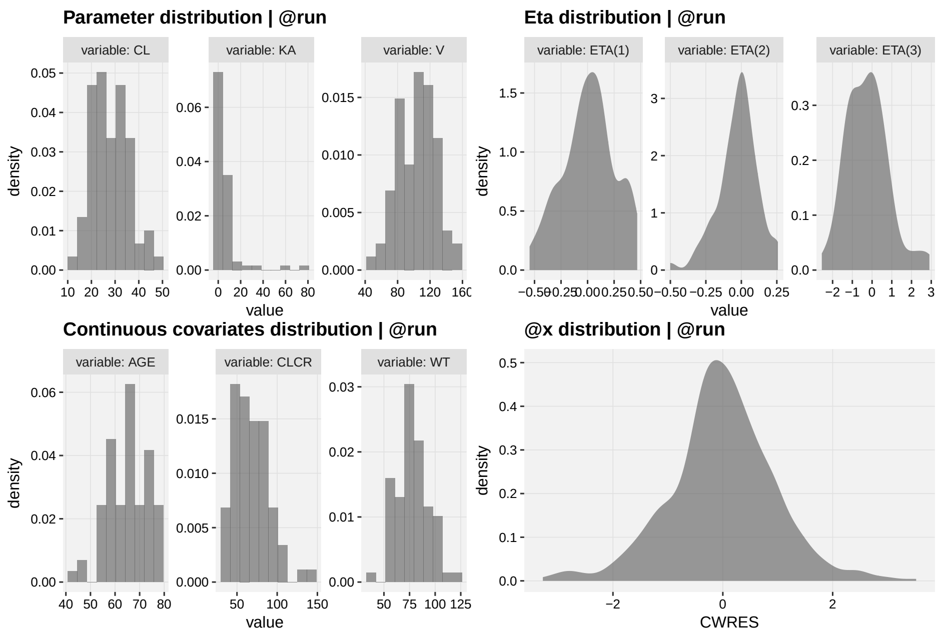

```
dv_preds_vs_idv(xpdb)
```



4 Distributions

Use `xxx_distrib()` to create distribution plots for `parameters`, `etas`, `covariates` and `residuals`. Use the `type=` to change plot type from histogram `h` to desity `d`.

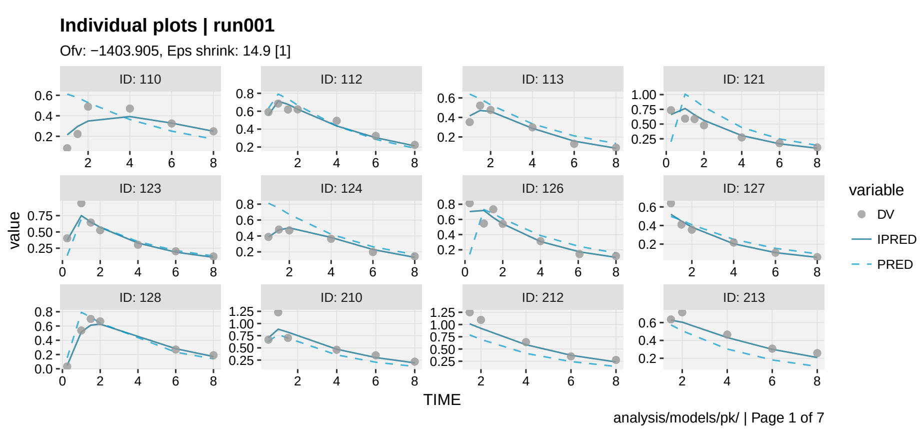
```
gridExtra::grid.arrange(
  prm_distrib(xpdb, subtitle = NULL, caption = NULL, type = 'h'),
  eta_distrib(xpdb, subtitle = NULL, caption = NULL, type = 'd'),
  cov_distrib(xpdb, subtitle = NULL, caption = NULL, type = 'h'),
  res_distrib(xpdb,res = "CWRES",subtitle = NULL, caption = NULL, type = 'd'),
  ncol=2)
```



5 Individual plots

The `ind_plots` plots the individually faceted fits.

```
ind_plots(xpdb, page = 1,
          ncol = 4, nrow = 3)
```



6 Visual Predictive Checks

6.1 Introduction

VPC can be created either by:

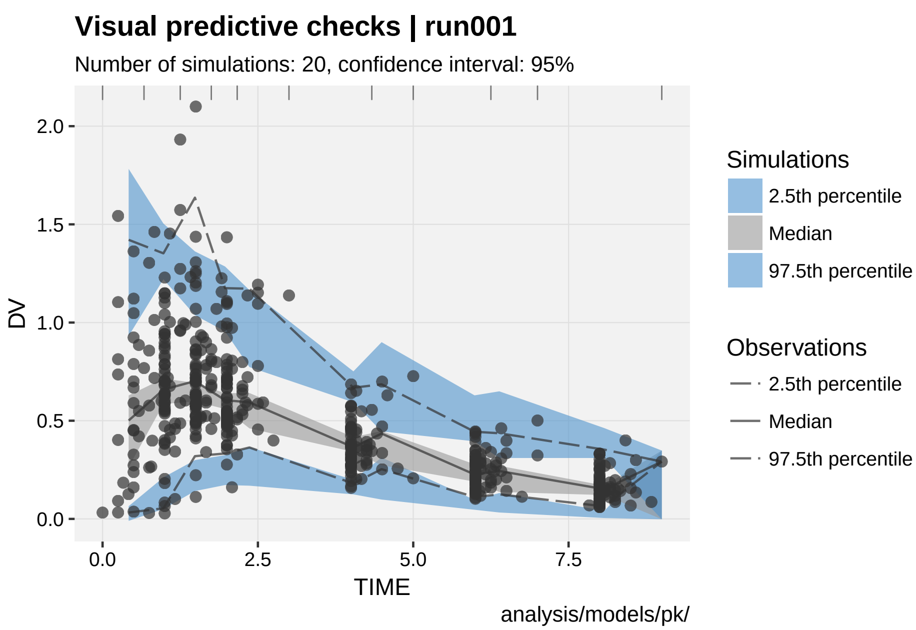
1. Using an xpdb containing a simulation and an estimation problem
2. Using a PsN generated VPC folder

The VPC functionality in xpose is build around the vpc R package. For more details about the way the vpc package works, please check the documentation website.

6.2 Basics of VPC in xpose

The VPC computing and plotting parts have been separated into two distinct functions: `vpc_data()` and `vpc()` respectively.

```
xpdb %>%
 vpc_data() %>%
  vpc()
```



6.3 Creating VPC using the xpdb data

To create VPC using the xpdb data, at least one simulation and one estimation problem need to present. Hence in the case of NONMEM the run used to generate the xpdb should contain several `$PROBLEM`. In `vpc_data()` the problem number can be specified for the observation (`obs_problem`) and the simulation (`sim_problem`). By default xpose picks the last one of each to generate the VPC.

6.4 Creating the VPC using a PsN folder

The `vpc_data()` contains an argument `psn_foler` which can be used to point to a PsN generated VPC folder. As in most xpose function `template_titles` keywords can be used to automatize the process e.g.

`psn_folder = '@dir/@run_vpc'` where `@dir` and `@run` will be automatically translated to initial (i.e. when the xpdb was generated) run directory and run number `'analysis/models/pk/run001_vpc'`.

In this case, the data will be read from the `/m1` sub-folder (or `m1.zip` if compressed). Note that PsN drops unused columns to reduce the simtab file size. Thus, in order to allow for more flexibility in R, it is recommended to use multiple stratifying variables (`-stratify_on=VAR1,VAR2`) and the prediction corrected (`-predcorr` adds the PRED column to the output) options in PsN to avoid having to rerun PsN to add these variables later on. In addition, `-dv`, `-idv`, `-lloq`, `-uloq`, `-predcorr` and `-stratify_on` PsN options are automatically applied to xpose VPC.

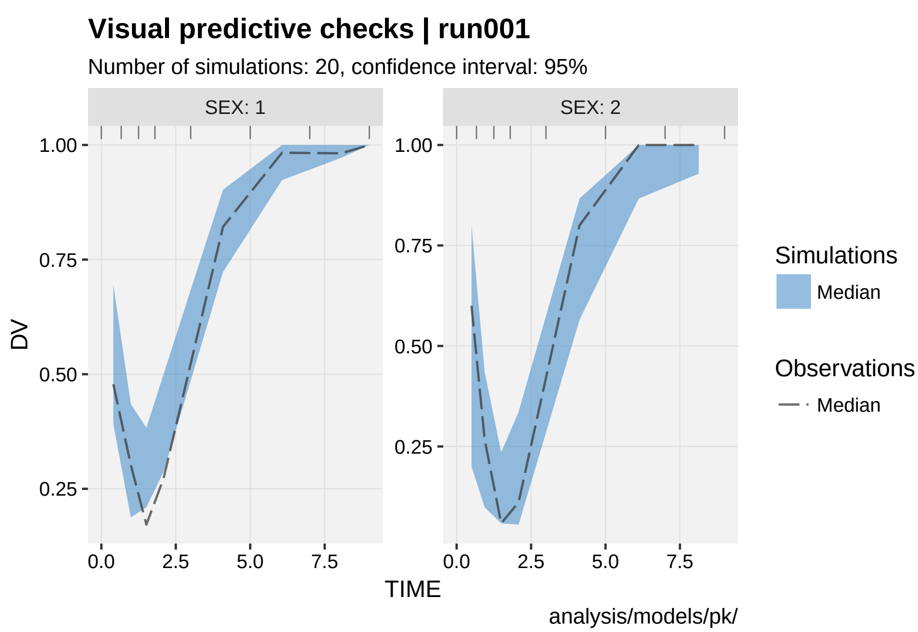
The PsN generated binning can also applied to xpose VPC with the `vpc_data()` option `psn_bins = TRUE` (disabled by default). However PsN and the vpc package work slightly differently so the results may not be optimal and the output should be evaluated carefully.

```
xpdb %>%
 vpc_data(psn_folder = '@dir/run001_vpc', psn_bins = TRUE) %>%
 vpc()
```

6.5 Options in `vpc_data()`

- The option `vpc_type` allows to specify the type of VPC to be computed: "continuous" (default), "categorical", "censored", "time-to-event".
- The `stratify` options defines up to two stratifying variable to be used when computing the VPC data. The `stratify` variables can either be provided as a character vector (`stratify = c('SEX', 'MED1')`) or a formula (`stratify = SEX~MED1`) . The former will result in the use of `ggforce::facet_wrap_paginate()` and the latter of `ggforce::facet_grid_paginate()` when creating the plot. With "categorical" VPC the "group" variable will also be added by default.
- More advanced options (i.e. binning, pi, ci, predcorr, lloq, etc.) are accessible via the `opt` argument. The `opt` argument expects the output from the `vpc_opt()` functions argument.

```
xpdb %>%
 vpc_data(vpc_type = 'censored', stratify = 'SEX',
          opt = vpc_opt(bins = 'jenks', n_bins = 7, lloq = 0.5)) %>%
 vpc()
```



6.6 Options in vpc()

- The option `vpc_type` works similarly to `vpc_data()` and is only required if several VPC data are associated with the xpdb.
- The option `smooth = TRUE/FALSE` allows to switch between smooth and squared shaded areas.
- The plot VPC function works similarly to all other xpose functions to map and customize aesthetics. However in this case the `area_fill` and `line_linetype` each require three values for the low, median and high percentiles respectively.

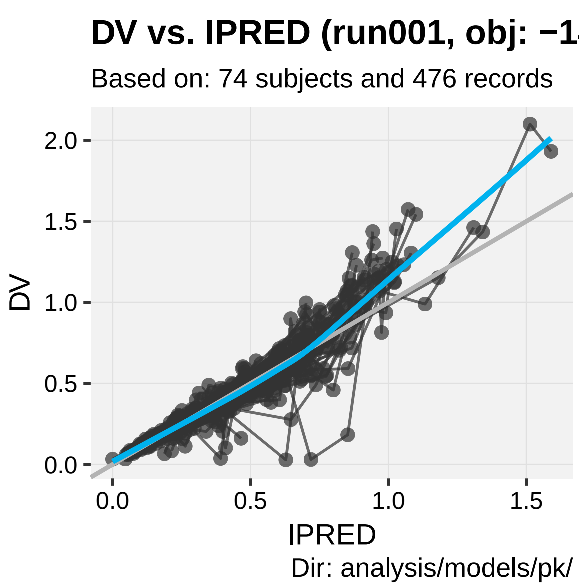
7 Cutomize plots

7.1 Labels

All xpose plots have by default an informative title, subtitle and caption. For example all plots using individual model predictions (IPRED) will display the epsilons' shrinkage. These titles can easily be edited as templates using @keywords which will be replaced by their actual value stored in the summary level of the `xpdb` object when rendering the plots. Keywords are defined by a word preceded by a `@` e.g. `'@ofv'`. A list of all available keyword can be accessed via `help('template_titles')`. The title, subtitle or caption can be disabled by setting them to `NULL`. Suffix can be automatically added to title, subtitle and caption of all plots. The suffixes can be defined in the `xp_theme`.

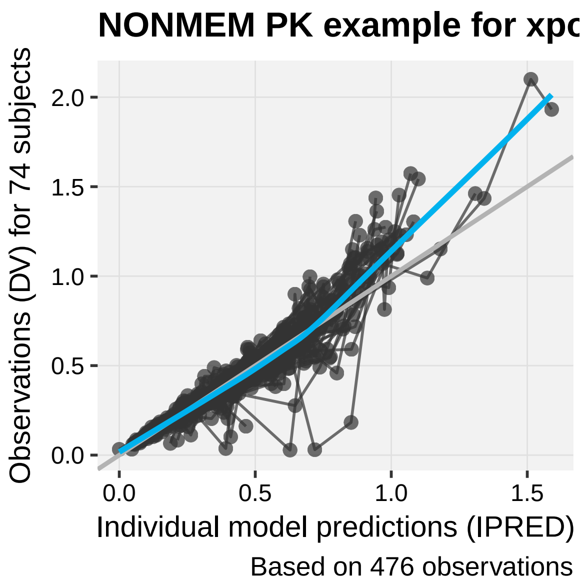
There are two ways to go about this:

```
# Using xpose
dv_vs_ipred(xpdb,
            title    = '@y vs. @x (@run, obj: @ofv)',
            subtitle = 'Based on: @nind subjects and @nobs records',
            caption  = 'Dir: @dir')
```




```
# Using ggplot
dv_vs_ipred(xpdb) +
  labs(title    = '@descr',
       subtitle = NULL,
       caption  = 'Based on @nobs observations',
```

```
       x         = 'Individual model predictions (@x)',
       y         = 'Observations (@y) for @nind subjects')
```

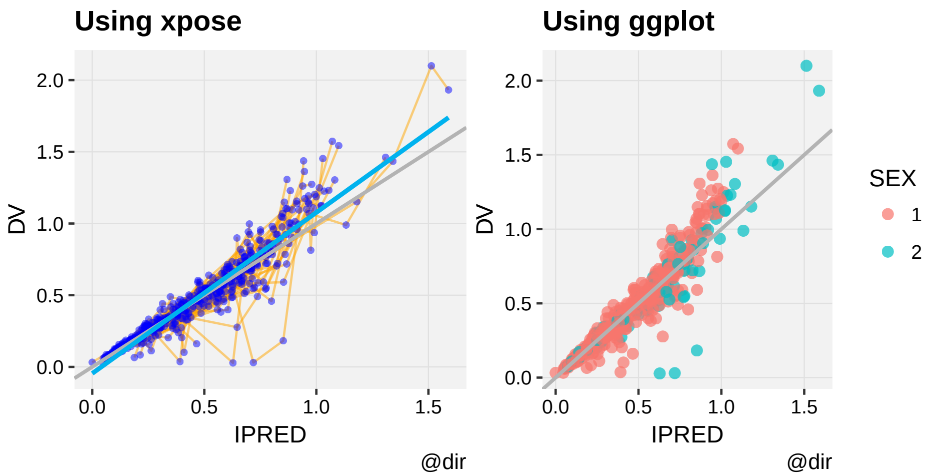


7.2 Modify aesthetics

By default the aesthetics are read from the `xp_theme` level in the xpdb object but these can be modified in any plot function. xpose makes use of the `ggplot2` functions mapping for any layer (e.g. points, lines, etc.) however to direct the mapping to a specific layer, a prefix appealing to the targeted layer should be used. The format is defined as `layer_aesthetic = value`. Hence to change the color of points in ggplot2 the argument `color = 'green'` could be used in `geom_point()`, while in xpose the same could be achieved with `point_color = 'green'`.

In basic goodness-of-fit plots, the layers have been named as: `point_xxx`, `line_xxx`, `smooth_xxx`, `guide_xxx`, `xscale_xxx`, `yscale_xxx` where `xxx` can be any option available in the `ggplot2` layers: `geom_point`, `geom_line`, `geom_smooth`, `geom_abline`, `scale_x_continuous`, etc.

```
#Using xpose
a <- dv_vs_ipred(xpdb,
            title = "Using xpose", subtitle = NULL,
            # Change points aesthetics
            point_color = 'blue', point_alpha = 0.5,
            point_stroke = 0, point_size = 1.5,
            # Change lines aesthetics
            line_alpha = 0.5, line_size = 0.5,
            line_color = 'orange', line_linetype = 'solid',
            # Change smooth aesthetics
            smooth_method = 'lm')

#Using ggplot
b <- dv_vs_ipred(xpdb,
                 type = 'p',title = "Using ggplot", subtitle = NULL,
                 aes(point_color = SEX))

gridExtra::grid.arrange(a,b,ncol=2)
```

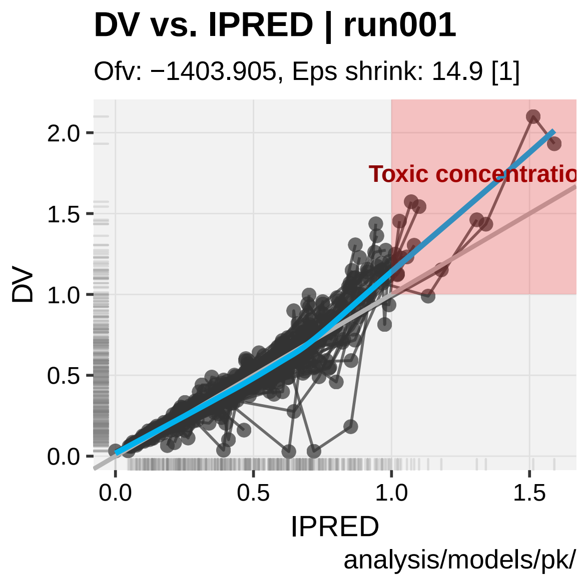


7.3 Additional layers

`xpose` offers the opportunity to add any additional layers from `ggplot2`. Example, a `ggplot2::geom_rug()` layer could be added to the `dv_vs_ipred()` plot along with some annotations (`ggplot2::annotate()`). Note: the additional layers do not inherit from the xpose aesthetic mapping (i.e. colors or other options need to be defined in each layer as shown below).

Layers can also be used to modify the aesthetics scales for example `ggplot2::scale_color_manual()`, or remove a legend `ggplot2::scale_fill_identity()`.

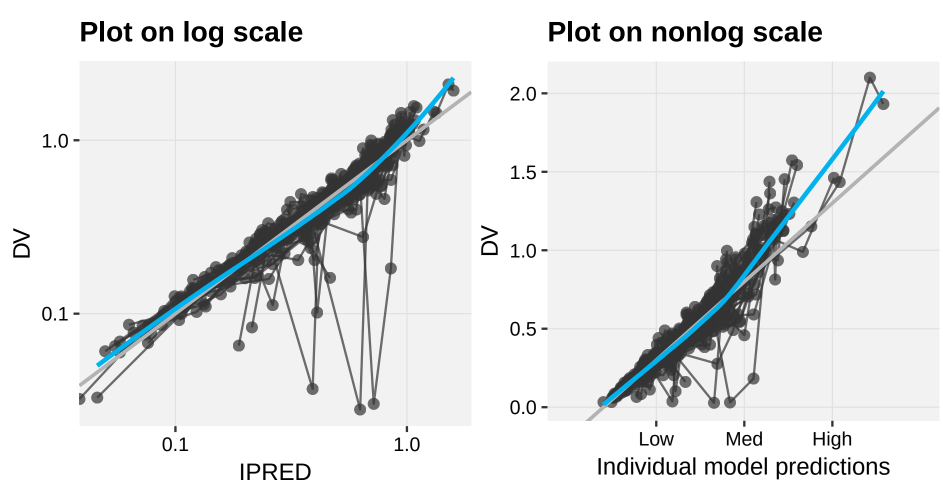
```
dv_vs_ipred(xpdb) +
  geom_rug(alpha = 0.2, color = 'grey50',
           sides = 'lb', size = 0.4) +
  annotate(geom = 'text',
           fontface = 'bold',
           color = 'darkred',
           size  = 3,
           label = 'Toxic concentrations', x = 1.35, y = 1.75) +
  annotate(geom = 'rect',
           alpha = 0.2, fill = 'red',
           xmin = 1, xmax = Inf,
           ymin = 1, ymax = Inf)
```



7.4 Scales

The argument `log` allows to log-transform the axes. Accepted values are `x`, `y` or `xy`. Additional arguments can be provided to the scales via the mapping by using the naming convention `xscale_xxx` or `yscale_xxx` where `xxx` is the name of a `ggplot2` scale argument such as `name`, `breaks`, `labels`, `expand`.

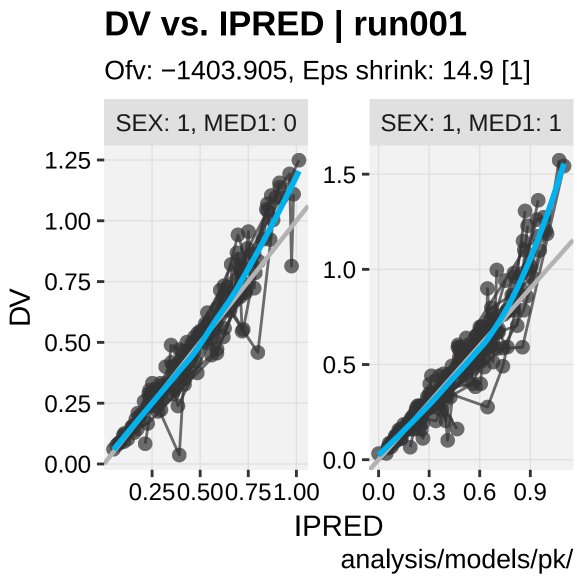
```
a <- dv_vs_ipred(xpdb, log = 'xy',
                 title = 'Plot on log scale',
                 subtitle = NULL,caption=NULL)

b <- dv_vs_ipred(xpdb,
            title = 'Plot on nonlog scale',
            subtitle = NULL, caption = NULL,
            xscale_breaks = c(0.3, 0.8, 1.3),
            xscale_labels = c('Low', 'Med', 'High'),
            xscale_expand = c(0.2, 0),
            xscale_name = 'Individual model predictions')

gridExtra::grid.arrange(a,b,ncol=2)
```



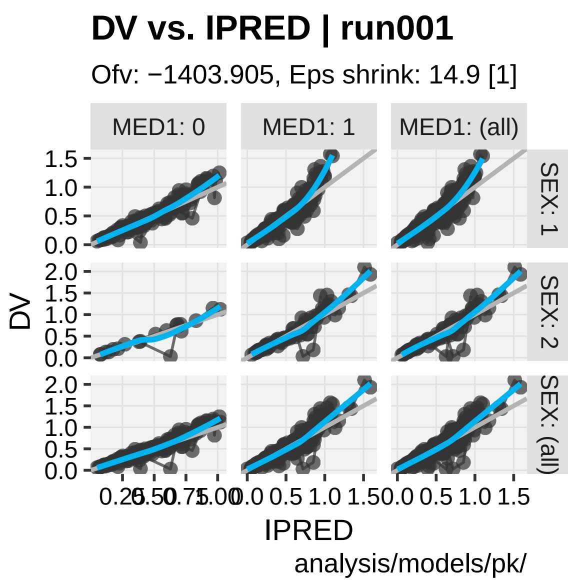
7.5 Facets

Panels (or faceting) can be created by using the `facets` argument as follows:

```r
# Example with a string
dv_vs_ipred(xpdb, facets = c('SEX', 'MED1'), ncol = 2, nrow = 1, page = 1)
```




```r
# Example with a formula
dv_vs_ipred(xpdb, facets = SEX~MED1, margins = TRUE)
```



Session 6: Looping Structures

Samuel P Callisto

July 19, 2018

Contents

Part 2: Looping Structures

Great reference: 'R for Data Science: Iteration'

Iteration is an important aspect of coding because it allows you to repeat operations multiple times without copy-pasting sections of your code. There are many looping structures available in R, but the most commonly used is the for-loop.

For-loops

Three essential components for a for-loop

- Output
- Sequence
- Body

```
## create example dataset
df <- data.frame(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

output <- vector("double", 4)       # 1. output
for (i in 1:ncol(df)) {             # 2. sequence
  output[i] <- median(df[,i])       # 3. body
}
output
```

```
## [1] -0.11937503 -0.08361914  0.40557951  0.23181049
```

About the output container

It may be tempting to skip this step and grow the size of your output variable with each iteration. However, this uses much more memory and can cause your computer to crash on very large datasets ("computationally expensive"). Best practice is to always create a container for your output prior to generating it, such as a vector, matrix, list, or data.frame. If you are unable to know the length of your output before starting the loop, save your output in a list data type (see 'R for Data Science: Unknown Output Length').

Slightly fancier method

- calculating number of required spaces from dataset rather than hard-coding 4; this allows for more flexible input
- seq_along() is a wrapper function for length() which avoids zero-length vector errors
- can use element selection [[]] for both vectors and data.frames rather than using different syntax

```
output <- vector("double", ncol(df))  # 1. output
for (i in seq_along(df)) {            # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
```

```
## [1] -0.11937503 -0.08361914  0.40557951  0.23181049
```

Exercise 1

Create a for-loop that iterates through the mtcars dataset and calculates the mean for each column

Exercise 2

Generate 10 random normals from each of μ = -10, 0, 10, 100

Alternate for-loop structures

The basic for-loop uses the variable i as an index through a vector or data.frame. In some cases it might be advantageous to access values directly rather than by using an index. There are two alternate methods for iteration using for-loops: 1: n in names(xs) 2: x in xs

```
## create empty vector for output with meaningful name
mtcarsMeans <- vector("numeric", ncol(mtcars))

## add column names to output vector
names(mtcarsMeans) <- names(mtcars)

## loop through columns to calculate mean for each
for(i in names(mtcars)){
  mtcarsMeans[[i]] <- mean(mtcars[[i]])
}

## display output
mtcarsMeans
```

```
##        mpg        cyl       disp         hp       drat         wt
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
##       qsec         vs         am       gear       carb
##  17.848750   0.437500   0.406250   3.687500   2.812500
```

Breaking out of loops

Break

Sometimes there will be a case in which you want to stop (break) or skip (next) when you encounter an element.

```
lettersBeforeP <- ""                                 # output
for(i in LETTERS){                                   # sequence
  lettersBeforeP <- paste(lettersBeforeP,i,sep= " ")  # body
  if(i == "P"){
    break
  }
}
lettersBeforeP
```

```
## [1] " A B C D E F G H I J K L M N O P"
```

Next

Using break will cause the loop to end, but you can use next to skip to the next iteration and continue looping

```
alphaWithoutSAM <- ""                              # output
for(i in letters){                                 # sequence
  if(i == "s" | i == "a" | i == "m") next        # body
  alphaWithoutSAM <- paste(alphaWithoutSAM, i, " ")
}
alphaWithoutSAM
```

```
## [1] " b   c   d   e   f   g   h   i   j   k   l   n   o   p   q   r   t   u   v   w   x   y   z   "
```

Nesting for-loops

Sometimes you will be utilizing multi-level data that varies by multiple factors. In these cases we can utilize multiple for-loops nested within each other.

Example: creating a times table from one through ten

```
## create output container
timesTable <- matrix(-99,nrow=10, ncol=10)

## iterate through row dimension
for(i in 1:10){
  ## iterate through column dimension
  for(j in 1:10){
    timesTable[i,j] = i*j
  }
}
timesTable
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    2    3    4    5    6    7    8    9    10
##  [2,]    2    4    6    8   10   12   14   16   18    20
##  [3,]    3    6    9   12   15   18   21   24   27    30
##  [4,]    4    8   12   16   20   24   28   32   36    40
##  [5,]    5   10   15   20   25   30   35   40   45    50
##  [6,]    6   12   18   24   30   36   42   48   54    60
##  [7,]    7   14   21   28   35   42   49   56   63    70
##  [8,]    8   16   24   32   40   48   56   64   72    80
##  [9,]    9   18   27   36   45   54   63   72   81    90
## [10,]   10   20   30   40   50   60   70   80   90   100
```

Example: loading multiple files into R

Say we have 8 files with a consistent naming pattern: 01_session_TPM.csv 01_session_PBO.csv 02_session_TPM.csv 02_session_PBO.csv 04_session_TPM.csv 04_session_PBO.csv 07_session_TPM.csv 07_session_PBO.csv We want to import all these files to analyze. Since the files are varying by two dimensions, we can use nested for-loops to generate all possible combinations.

```
## combinations of subject IDs and drugs
SID <- c("01", "02", "04", "07")
```

```
drug <- c("TPM", "PBO")

## output container
allSubjectData <- list()

## error catching statement
try(
## sequence through each subject ID
for(i in SID){
  ## sequence through each drug
  for(j in drug){
    ## generate a name for each element of the list
    combo <- paste0(i,"-",j)
    ## import files into the named element of the list
    allSubjectData[[combo]] <- read.csv(paste0(i,"_session_",j,".csv"))
  }
}
, silent = T)
```

```
## Warning in file(file, "rt"): cannot open file '01_session_TPM.csv': No such
## file or directory
```

While-loops

While

This type of looping structure is useful for situations when the number of iterations necessary for the task to finish is unknown. It should be noted that all for-loops can be re-written as as a while-loop, but the opposite is not true.

```
countToTen <- ""
i <- 1
while(i <= 10){
  countToTen <- paste(countToTen, i, sep=" ")
  i <- i + 1
}
countToTen
```

```
## [1] " 1 2 3 4 5 6 7 8 9 10"
```

Repeat

A modifed version of the while loop is repeat, which will keep running until it reaches a break statement.

```
countToTen <- ""
i <- 1
repeat{
  countToTen <- paste(countToTen, i, sep=" ")
  if(i==10){
    break
  }else{
    i <- i + 1
  }
```

```
}
countToTen
```

```
## [1] " 1 2 3 4 5 6 7 8 9 10"
```

Exercise 3

Calculate the mean for all columns in iris that contain numeric data using a while-loop

Apply and Purrr

Base R gives us the apply() family of functions, which can be useful alternatives to for-loops. Let's revisit the example from earlier of calculating the mean for all columns in the mtcars dataset

```
apply(mtcars,2,mean)
```

```
##        mpg        cyl       disp         hp       drat         wt
##  20.090625   6.187500 230.721875 146.687500   3.596563   3.217250
##       qsec         vs         am       gear       carb
##  17.848750   0.437500   0.406250   3.687500   2.812500
```

If your function can easily adhere to the syntax of the apply() functions, it can give the same results as a for-loop much less code. * apply(): evaluate a function over the margins of a matrix or array * lapply(): evaluate a function on each element in a list, and return the results as a list * sapply(): evaluate a function on each element in a list, and return the results in a "simplified form" (not always predictable, but can be convenient) * vapply(): similar to sapply(), but with more consistent return types * tapply(): evaluate a function on subsets of a vector; alternative to group_by() for dealing with subsets

Another useful package in the tidyverse is 'purrr', which is similar to apply, but strives to be more consistent with argument structure and syntax.

The difference in run-time between all these different methods of iteration is fairly similar (though purrr is likely the fastest), so just choose whichever syntax you prefer.

Session 7: An Introduction to Shiny

Dave Margraf

July 26, 2018

Contents

The `shiny` package

- Build interactive web apps using R.

Look at some published examples.

Download the package if needed:

```
install.packages("shiny")
```

List examples provided in the package with 'runExample()'.

```
## Valid examples are "01_hello", "02_text", "03_reactivity", "04_mpg", "05_sliders", "06_tabsets", "07_
```

Run a local example.

- Open a new R script (Ctrl + Shift + N) and paste the following code:

```
library(shiny)
runExample("05_sliders")
```

Copying and modifying existing apps is a useful way to get used to the structure and functionality of Shiny.

```r
runExample("01_hello")      # a histogram
runExample("02_text")       # tables and data frames
runExample("03_reactivity") # a reactive expression
runExample("04_mpg")        # global variables
runExample("05_sliders")    # slider bars
runExample("06_tabsets")    # tabbed panels
runExample("07_widgets")    # help text and submit buttons
runExample("08_html")       # Shiny app built from HTML
runExample("09_upload")     # file upload wizard
runExample("10_download")   # file download wizard
runExample("11_timer")      # an automated timer
```

Shiny components

- A Shiny app requires a user interface, a server function, and a call to the 'shinyApp()' function.
- The user interface
  - Defines inputs
  - Inputs and outputs laid out
- Server function
  - Creates output and other data
- These may be in separate files or together.

A basic single file template:

```r
library(shiny)

ui <- fluidPage()

server <- function(input, output){}

shinyApp(ui = ui, server = server)
```

Open the R file named 'shinySingleFile.R' if you would like to build off of it.

User interface

- Use the 'fluidPage()' function to define the layout of your app.
- Section of the page are broken up into panels or rows based on an underlying grid.
- The sidebar and grid layouts are common and easy to begin with.

Sidebar Layout

- Provides a sidebar for inputs and a large main area for output.

```r
ui <- fluidPage(

  titlePanel("Sidebar layout example"),

  sidebarLayout(

    sidebarPanel(
      # Add widgets here
    ),

    mainPanel(
      # Add plots here
    )
  )
))
```

Grid layout

- Rows are created by the 'fluidRow()' function.
- Columns defined by the column() function.
- Column widths are 12-wide grid system within a 'fluidRow()'.

```r
ui <- fluidPage(

  titlePanel("Grid layout example"),

  fluidRow(
    column(4,
    ),
    column(4,
    ),
    column(4,
    )
  ),

  fluidRow(
    column(2,
    ),
    column(6,
    ),
    column(4,
    )
  )
)
```

Control widgets

- Widgets add web elements to your Shiny app.
- Users can interact with widgets provide values to Shiny.

Standard control widgets:

Function	Widget
‘actionButton()’	Action Button
‘checkboxGroupInput()’	A group of check boxes
‘checkboxInput()’	A single check box
‘dateInput()’	A calendar to aid date selection
‘dateRangeInput()’	A pair of calendars for selecting a date range
‘fileInput()’	A file upload control wizard
‘helpText()’	Help text that can be added to an input form
‘numericInput()’	A field to enter numbers
‘radioButtons()’	A set of radio buttons
‘selectInput()’	A box with choices to select from
‘sliderInput()’	A slider bar
‘submitButton()’	A submit button
‘textInput()’	A field to enter text

Open ‘widgetGallery.R’ to test some of these and press ‘Run App’.

Server function

The ‘server()’ builds a list-like object named output that contains all of the code needed to update the ‘R’ objects in your app. Each R object needs to have its own entry in the list.

Display reactive output

You can create reactive output by adding an object to your user interface and telling ‘R’ to build the object in the server function.

```
server <- function(input, output){}
```

Reactivity is acheived connecting the widget values (the source) of ‘input’ to the objects (the endpoints) in ‘output’ in the above code.

Output function	Creates
dataTableOutput	DataTable
htmlOutput	raw HTML
imageOutput	image
plotOutput	plot
tableOutput	table
textOutput	text
uiOutput	raw HTML
verbatimTextOutput	text

Add output to the user interface ‘sidebarPanel()’, ‘mainPanel()’, or ‘column()’ in the ‘ui’ to tell Shiny where to display your object.

Next, provide code to build the object in the ‘server()’ function.

Render function	Creates
renderDataTable	DataTable

Render function	Creates
renderImage	images (saved as a link to a source file)
renderPlot	plots
renderPrint	any printed output
renderTable	data frame, matrix, other table like structures
renderText	character strings
renderUI	a Shiny tag object or HTML

Create an entry in the server output list by defining a new element within the 'server()' function. The element name should match the name of the reactive element that you created in the 'ui'.

For example, the element 'output$selectOut' is defined by the 'renderPrint()' function in the 'server()' function and matches names with 'verbatimTextOutput("selectOut")' of the 'ui' in the 'widgetGallery.R' file. If you can understand that gibberish you're well on your way to understaning Shiny.

Let's take a look at a simple example:

- Open a new R script (Ctrl + Shift + N) and paste the following code:

```r
library(shiny)
runExample("01_hello")
```

The reactive source, 'input$obs', is used by the reactive endpoint, 'output$distPlot'. Whenever 'input$obs' changes, 'output$distPlot' is notified that it needs to re-execute.

Session 8: Function Writing

Samuel P Callisto

August 2, 2018

Contents

Function Writing

A trivial example

```
sumMinusOne <- function(x){
  output <- 0
  for(i in 1:length(x)){
    output <- output + x[i]
  }
  return(output-1)
}

sumMinusOne(2:4)
```

```
## [1] 8
```

When would I ever use this?

Example 1: Wrapper function

Problem: importing files with multiple headers causes the data type to be interpretted incorrectly, causing resulting in manual typecasting for multiple rows (annoying!)

```
## Excel files
excel <- read.csv("datasets/TPM_sim_dataset_20180607.csv", as.is = T, header = T)

## dataset imported using read.csv()
str(excel)
```

```
## 'data.frame':    30 obs. of  5 variables:
##  $ subjectid: chr  "SID" "3" "4" "6" ...
##  $ DATE     : chr  "m/d/y" "5/4/2018" "5/5/2018" "5/6/2018" ...
##  $ TIME     : chr  "hh:mm" "5:40" "5:41" "5:42" ...
##  $ DV       : chr  "ug/mL" "0.156" "0.157" "-99" ...
##  $ SEX      : chr  "M=male" "M" "M" "F" ...
```

```
excel$subjectid <- as.integer(excel$subjectid)
excel$DV <- as.double(excel$DV)
```

Solution: write a wrapper function that assigns correct header row while maintaining data types.

```
headr <- function(file, header.row=1, data.start=3){
  headers <- read.csv(file = file, skip=header.row-1, header = F, nrows = 1, as.is = T)
  dataset <- read.csv(file=file, skip = data.start-1, header = F, as.is=T)
  names(dataset) <- headers
  return(dataset)
}

## import same file using headr wrapper function
topiramateData <- headr("datasets/TPM_sim_dataset_20180607.csv")

## dataset imported using headr()
str(topiramateData)
```

```
## 'data.frame':    29 obs. of  5 variables:
```

```
##  $ subjectid: int  3 4 6 7 11 16 35 38 39 40 ...
##  $ DATE     : chr  "5/4/2018" "5/5/2018" "5/6/2018" "5/7/2018" ...
##  $ TIME     : chr  "5:40" "5:41" "5:42" "5:43" ...
##  $ DV       : num  0.156 0.157 -99 0.159 0.16 0.161 0.162 0.163 0.164 0.165 ...
##  $ SEX      : chr  "M" "M" "F" "M" ...
```

Notice how few arguments need to be filled out manually each time you import a file using the helper function since you are allowed to set your defaults.

Slightly more advanced solution: You can use the ellipsis operator (...) to pass additional commands into the functions called by your wrapper function. In this example, by adding this to headr(), we can access arguments in read.csv()

```
headr <- function(file, header.row=1, data.start=3, ...){
  headers <- read.csv(file = file, skip=header.row-1, header = F, nrows = 1, as.is = T)
  dataset <- read.csv(file=file, skip = data.start-1, header = F, as.is=T, ...)
  names(dataset) <- headers
  return(dataset)
}

## import same file using headr wrapper function
topiramateData <- headr("datasets/TPM_sim_dataset_20180607.csv", na.strings=-99)

## dataset imported using headr()
str(topiramateData)
```

```
## 'data.frame':    29 obs. of  5 variables:
##  $ subjectid: int  3 4 6 7 11 16 35 38 39 40 ...
##  $ DATE     : chr  "5/4/2018" "5/5/2018" "5/6/2018" "5/7/2018" ...
##  $ TIME     : chr  "5:40" "5:41" "5:42" "5:43" ...
##  $ DV       : num  0.156 0.157 NA 0.159 0.16 0.161 0.162 0.163 0.164 0.165 ...
##  $ SEX      : chr  "M" "M" "F" "M" ...
```

Anonymous Functions

R allows you to call a function without naming it, called an Anonymous Function. This is useful typically used when you are applying a small function to a matrix using apply().

```
apply(topiramateData, 2, range)
```

```
##      subjectid DATE         TIME   DV SEX
## [1,] " 3"      "2018-05-32" "5:40" NA "F"
## [2,] "73"      "5/9/2018"   "6:08" NA "M"
```

Due to the missing value in the DV column, we cannot get range values for this column. We need to use an anonymous function to pass the na.rm argument into the range function if we wish to apply it to the entire data.frame.

```
## this approach will only give us an error:
## apply(topiramateData, 2, range(na.rm=T))

## instead we can use an anonymous function to access arguments
apply(topiramateData, 2, function(x) range(x,na.rm=T))
```

```
##      subjectid DATE         TIME   DV      SEX
## [1,] " 3"      "2018-05-32" "5:40" "0.156" "F"
## [2,] "73"      "5/9/2018"   "6:08" "0.184" "M"
```

By creating an anonymous function, we can get results for all the columns in the data.frame. We can create any sort of anonymous function to use in this context; let's look at a more complicated example.

Example 2: Dealing with Times

Problem: RedCap stores my times as a character string ("6:45"), but I want to calculate the difference between observations.

Solution: write a function to use this time and the next time you encounter clock times

```
numericTime <- function(vec){
  ## separate hours and minutes into a vector
  sapply(strsplit(vec,":"),
         function(x) {
           ## convert to numeric type to allow arithmetic operations
           x <- as.numeric(x)
           ## numeric time = hours + (minutes/60) rounded to two decimals
           round(x[1]+x[2]/60,2)
         }
  )
}

topiramateData$DECTIME <- numericTime(topiramateData$TIME)
```

Challenge: Use RStudio's built-in debug functions to verify that x <- as.numeric(x) is necessary in our function.

A general rule of thumb:

If you find yourself copying code within or between files, you should probably just write a function. Better still, add functions to a personal R package that you can easily import and share.

Great resource for learning more about writing functions

http://adv-r.had.co.nz/Functional-programming.html

Unit testing

How can you check if your function is doing what you think it is? Let's go back to our sumMinusOne() function

A trivial example revisited

```
test_that("single value minus one",{
  expect_equal(1, sumMinusOne(2))
})

test_that("vector sum minus one",{
  expect_equal(5, sumMinusOne(1:3))
})

test_that("characters shouldn't work",{
  expect_error(sumMinusOne("test"))
  expect_error(sumMinusOne("1"))
})
```

There is no message if the test passes, but you will get an error if the tests fail. That means you can write multiple tests into your code and let them be tested automatically, alerting you if the change you just made altered the functionality in an unexpected way.

```
try(
test_that("characters shouldn't work",{
  expect_error(sumMinusOne(1))
})
, outFile = stdout())
```

```
## Error : Test failed: 'characters shouldn't work'
## * `sumMinusOne(1)` did not throw an error.
```

Testing our wrapper function for read.csv()

```
test_that("ID imported as integer",{
  imported <- headr("datasets/TPM_sim_dataset_20180607.csv")
  expect_true(is.integer(imported[1,1]))
})

test_that("time imported as character",{
  imported <- headr("datasets/TPM_sim_dataset_20180607.csv")
  expect_true(is.character(imported[1,3]))
})

test_that("concentrations imported as numeric",{
  imported <- headr("datasets/TPM_sim_dataset_20180607.csv")
  expect_true(is.numeric(imported[1,4]))
})
```

Testing numericTime()

```r
test_that("numbers shouldn't work",{
  expect_error(numericTime(5.37))
})

test_that("strings without colon shouldn't work",{
  expect_true(is.na(numericTime("546")))
  expect_true(is.na(numericTime("5.46")))
})

test_that("single digit & double digit times should work",{
  expect_equal(5.5, numericTime("5:30"))
  expect_equal(12.0, numericTime("12:00"))
  ## notice we don't test for real clock times
  ## we could add this into the function later
  expect_equal(25.75, numericTime("25:45"))
})

test_that("differences in time can now be calculated", {
  expect_equal(2, numericTime("5:00") - numericTime("3:00"))
  expect_equal(4.5, numericTime("4:30"), numericTime("12:00"))
})

test_that("vectorized application succeeds",{
  expect_equal(c(12, 1.5), numericTime(c("12:00", "1:30")))
})
```