PharmaSUG 2019 - Paper DV-169

The Power of Data Visualization in R

Babych Oleksandr, Experis Clinical

ABSTRACT

The ability to build beautiful and meaningful graphs is a valuable skill for the data analyst. R has become a popular programming language in the field of data analysis. Among many of its advantages, it has two essential ones: it is easy to learn and it has a powerful visualization package - ggplot2. Therefore, it wouldn't require a great amount of time learning how to make high-end graphs in R.

In this article we will take a look at how to build graphs via ggplot2 and we will consider its underlying concept called the Grammar of Graphics. The central idea of ggplot2 consists in constructing plots by combining different layers on top of each other and using aesthetic mappings to define the graph. The main objective of this paper is to highlight the advantages of this approach by providing various examples and to demonstrate how R can be a powerful tool in the skill set of any data analyst.

INTRODUCTION

The tidyverse is a consistent system of packages that covers the entire data analysis workflow. All these packages share a common design philosophy, grammar and data structures. In general, because of the connection between tools, the tidyverse makes data scientists and statisticians more productive and guides them through the workflow, as well as allows easier communication [1]. The core tidyverse contains 8 packages for data import (readr), tidying (tidyr, tibble), transforming or data manipulation (dplyr, stringr, forcats), data visualization (ggplot2) and functional programming (purrr). Most of these packages were developed by Hadley Wickham and in fact, ggplot was the first of them, which was later converted into ggplot2. ggplot2 was invented more than 10 years ago, so now despite the fact that R is an open-source language, ggplot2 itself can be changed insignificantly and mostly by adding new functions than by altering the old ones [2].

The grammar of graphics is a tool that allows us to briefly describe the components or so-called grammatical elements of a plot. This concept was suggested by Wilkinson, Anand, and Grossman (2005) and was later changed by Hadley Wickham by proposing an alternative parametrization of the grammar that was implemented in ggplot2 [3].

The basic idea of ggplot2 is as follows: a graph is made from layers that are combined in diverse ways to get the desired type of data visualization. There are three main layers – Data, Aesthetics and Geometries and four optional layers – Statistics, Facets, Coordinates, and Themes. The first three layers contain sufficient information to build a basic plot, but to make a more sophisticated and meaningful plot other layers have to be added.

In this paper, we will focus on visualizing data using ggplot2, and we will go through all the grammatical elements of a graph beginning with the first three layers that are essential for plotting graphs. Then we will discuss four optional layers and see how they can affect our data visualization and make it more advanced.

ESSENTIAL LAYERS OF GGPLOT2

1.1 FIRST STEPS ─ DATA AND AESTHETICS

Firstly, to start working with the ggplot2 package you need to install the core tidyverse — 8 packages that you will use in almost every data analysis. It is done by running the following line of code:

```
install.packages("tidyverse")
```

After that these packages need to be loaded into your working environment by running:

```
library(tidyverse)
```

The package must be installed once, but it always needs to be reloaded at the beginning of a new session [4]. The ggplot2 package can be installed and loaded separately from the whole tidyverse by running:

```
install.packages("ggplot2")
library(ggplot2)
```

But usually, it is more convenient to install the whole tidyverse, as tasks of the data analyst are not restricted only to the construction of graphs.

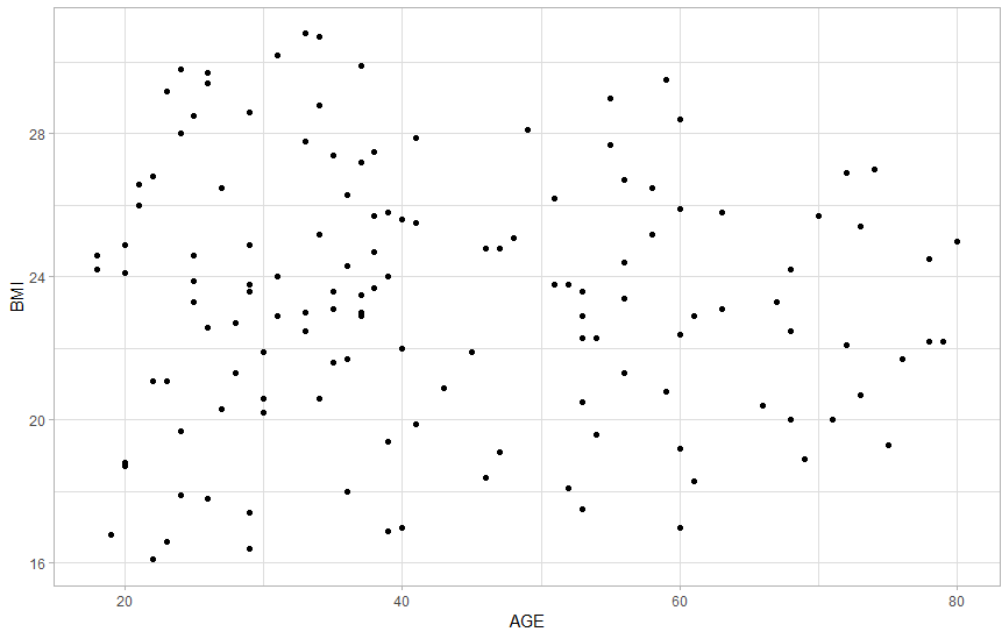
To demonstrate the capabilities of the package, two random data sets ASL and ALB were created. ASL data set containing 8 variables: USUSBJID, ARMCD, SEX, AGE, BMI, REGION, AGEGRP, and TRTDUR. There are totally 135 subjects divided into 3 different arms (A, B and C). And ALB data set containing: USUBJID (4 different USUBJIDs), ARMCD, SEX, AGEGRP, PARAMCD (only one PARAMCD = “ALBUMIN”), ADY, AVAL, AVALU, and AVISIT.

Initially, to produce a plot with ggplot2, we must define three main things:

1. A data frame containing the data.
2. Aesthetics – instructions on how the columns of the data frame must be converted into positions such as x and y-axis, colors, sizes, and shapes of graphical elements.
3. Geometrical objects - the actual graphical elements to display [5].

To plot the data frame we assign the name of our data frame to the argument *data*. After that we should map one variable on the x-axis and another variable on the y-axis inside the *aes( )* argument. And finally, we add the layer of points to our plot by setting *geom_point( )* function, which creates a scatterplot.

In Figure 1 we can see that x and y aesthetics are position of dots on the common scale where variable AGE is mapped onto x-axis and variable BMI is mapped onto y-axis.




```
ggplot(ASL, aes(x = AGE, y = BMI)) +
  geom_point()
```

Figure 1 - Basic scatter plot.

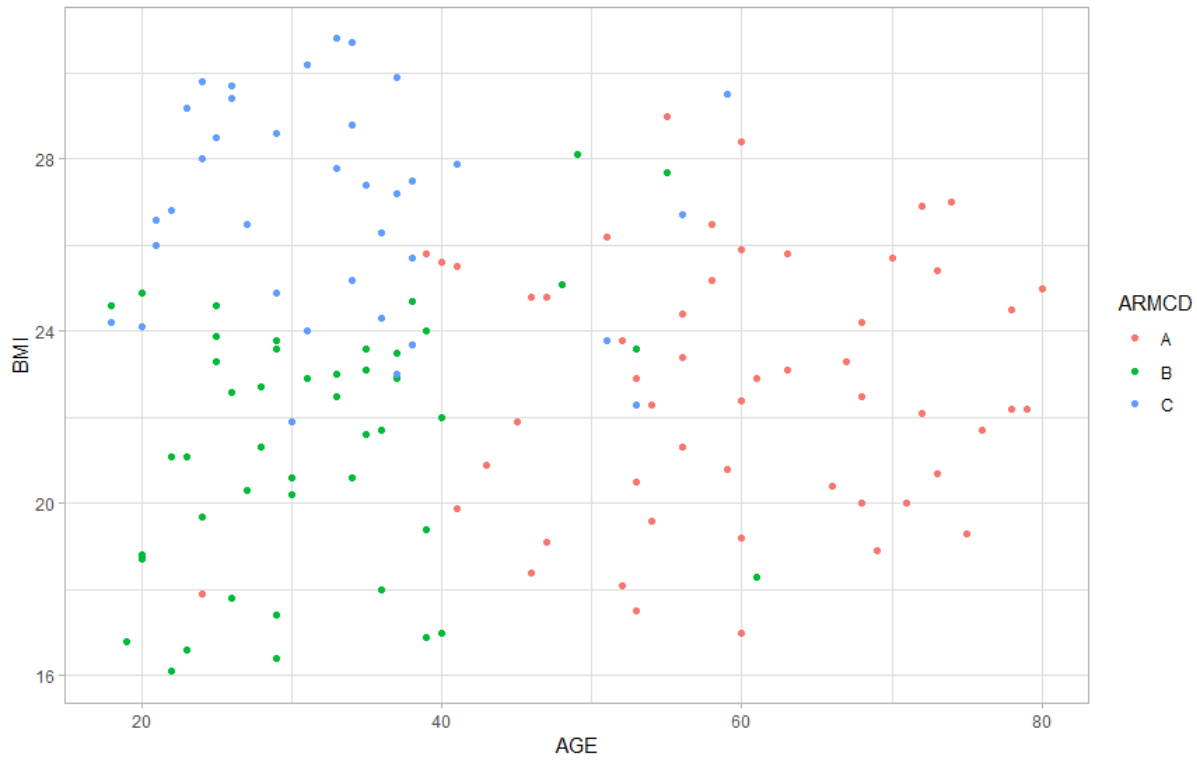
Note: you should not use the word “data” all the time inside the ggplot2, because it is a positional argument which is located on the first place, so we can just put the data frame’s name instead of the “data = ” (see Figure 2).

Let’s explore what aesthetics actually are. Commonly we believe that aesthetics describe some characteristics of the object like color, shape or size, but for ggplot2 these are attributes, which describe what something looks like. For example, if it is necessary to change the color of dots – we can do it in the geom layer by setting the *col* argument to “blue”:

```
geom_point(col = “blue”).
```

In this way we have changed the attribute of the dots, but not the aesthetics.

We already have two aesthetics on our plot, but let us increase their number by one more. As in ggplot2 the aesthetics are used to map variables, it is possible to add the third variable that can be mapped onto a visible aesthetic, for instance, *color*. That is how the color for the dots can be chosen according to the categorical variable ARMCD, as shown in Figure 2. This means that we can add the 3rd dimension or the 3rd scale to our plot, which is the color. So, aesthetics allow us to create multi-variable plots and increase the number of variables that can be shown on single data visualization.




```
ggplot(ASL, aes(x = AGE, y = BMI, col = ARMCD)) +
  geom_point()
```

Figure 2 - Scatter plot, where each point has a color in accordance with the category to which it belongs.

Figure 2 shows that we have mapped 3 variables of our data frame on 3 scales – x, y, and color. Also, it is easy to reveal that the legend was created automatically.

Usually, we call the aesthetic in the *aes( )* argument while attributes are called in the geom layer. However, aesthetics can be called in the geom layer, which gives exactly the same result (see Figure 7), but usually, it is done when different data frames need to be combined on the single plot.

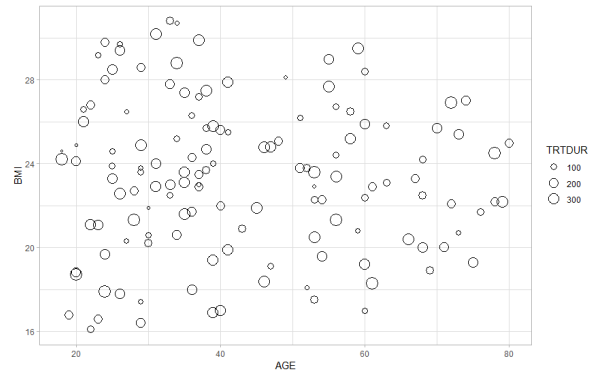
The most common aesthetics are shown in table 1.1

X	X-axis position
Y	Y-axis position
Color	Color of dots, outlines of other shapes
Size	Diameter of points, thickness of lines
Fill	Fill color
Alpha	Transparency
Shape	Shape
Linetype	Line dash pattern
Labels	Text on a plot or axis

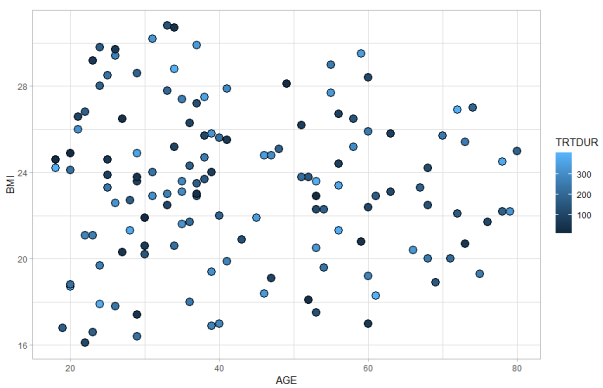
Table 1.1 Typical aesthetics.

Note: many of these aesthetics can be both aesthetic mappings as well as attributes. So, it is necessary to be very careful not to confuse or overwrite them. Also, lots of aesthetics can be mapped onto continuous and categorical variables, but labels and shapes can be mapped only on categorical (discrete) data.

Below is shown how the plot changes when the third variable is mapped to a different aesthetic. In Figure 3a the continuous variable TRTDUR is mapped onto the *size* aesthetic and also the *shape* attribute of the dots was changed. In Figure 3b TRTDUR is mapped onto the *fill* aesthetic and at the same time, we have changed the *size* and *shape* attributes. In Figure 4a we can see the *alpha* aesthetic in action and in Figure 4b categorical variable ARMCD is mapped onto *shape* aesthetics. So, in Figure 3a, Figure 3b and Figure 4a we can see how the size or the color of dots differs depending on the value of the continuous variable TRTDUR. And Figure 4b is pretty similar to Figure 2 except that in Figure 2 the dots have various colors and in Figure 4b – various shapes. You can see examples of different shape attributes on the Cookbook for R [6].

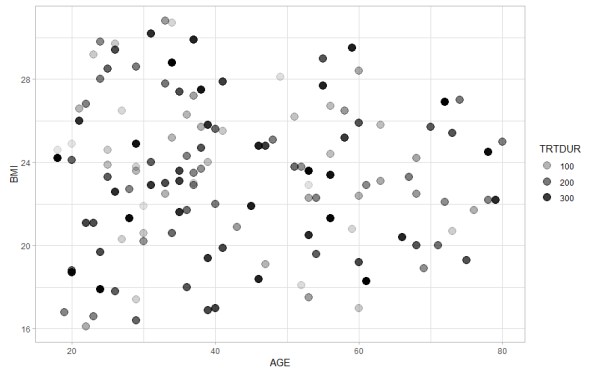



```
ggplot(ASL, aes(x = AGE, y = BMI,
  size = TRTDUR)) +
  geom_point(shape = 21)
```

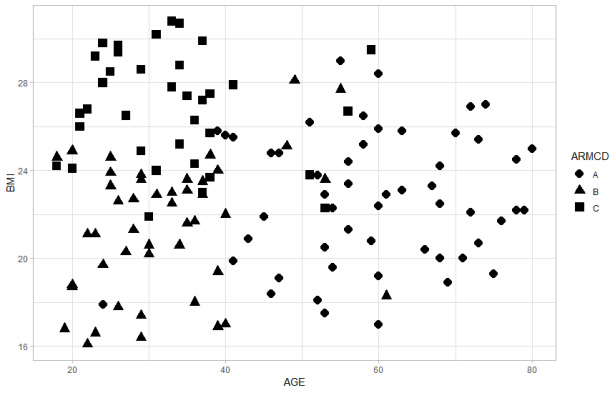



```
ggplot(ASL , aes(x = AGE, y = BMI,
  fill = TRTDUR)) +
  geom_point(shape = 21, size = 4)
```

Figure 3 - Size and fill aesthetics in action.



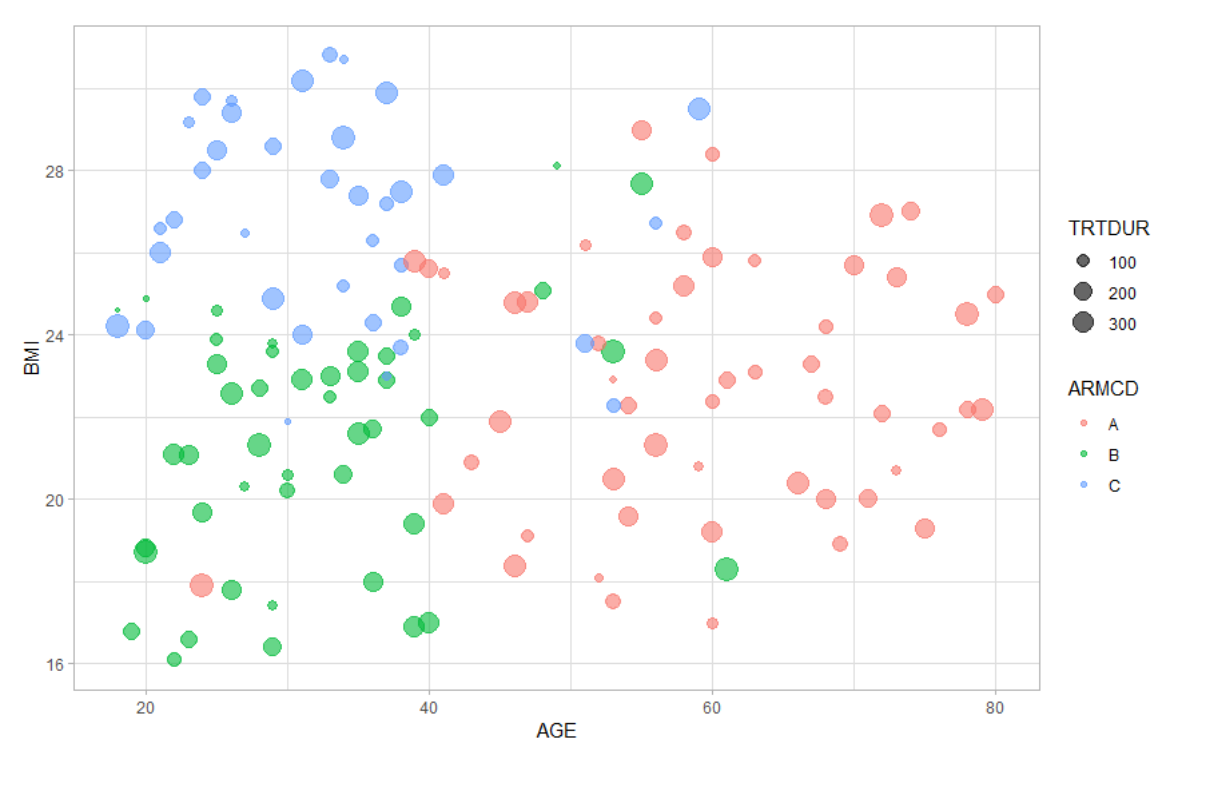

```
ggplot(ASL, aes(x = AGE, y = BMI,
  alpha = TRTDUR)) +
  geom_point(size = 4)
```




```
ggplot(ASL, aes(x = AGE, y = BMI,
  shape = ARMCD)) +
  geom_point(size = 4)
```

Figure 4 - Alpha and shape aesthetics in action.

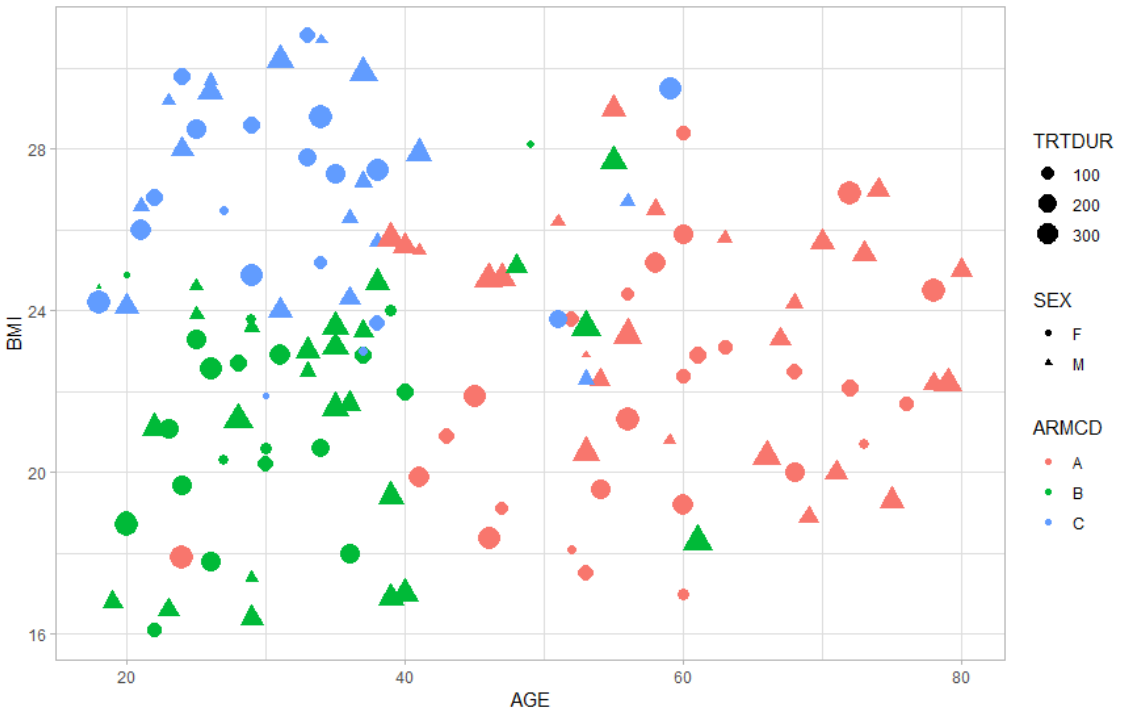
ggplot2 allows to add several optional aesthetics, so in Figure 5 we took the graph from Figure 2 and added the continuous variable TRTDUR by mapping it on the *size* aesthetic. Also, the alpha-blending attribute was changed by specifying the *alpha* argument inside the *geom_point( )* function.




```
ggplot(ASL, aes(x = AGE, y = BMI, col = ARMCD, size = TRTDUR)) +
   geom_point(alpha = 0.6)
```

Figure 5 - Four variables are mapped to different aesthetics.

In Figure 5 we've mapped 4 variables from the ASL data frame to 4 different aesthetics, but it is still not the final stage. Moreover, we can add the fifth variable to the plot from Figure 5 by mapping categorical variable SEX to the *shape* aesthetic, as shown in Figure 6.

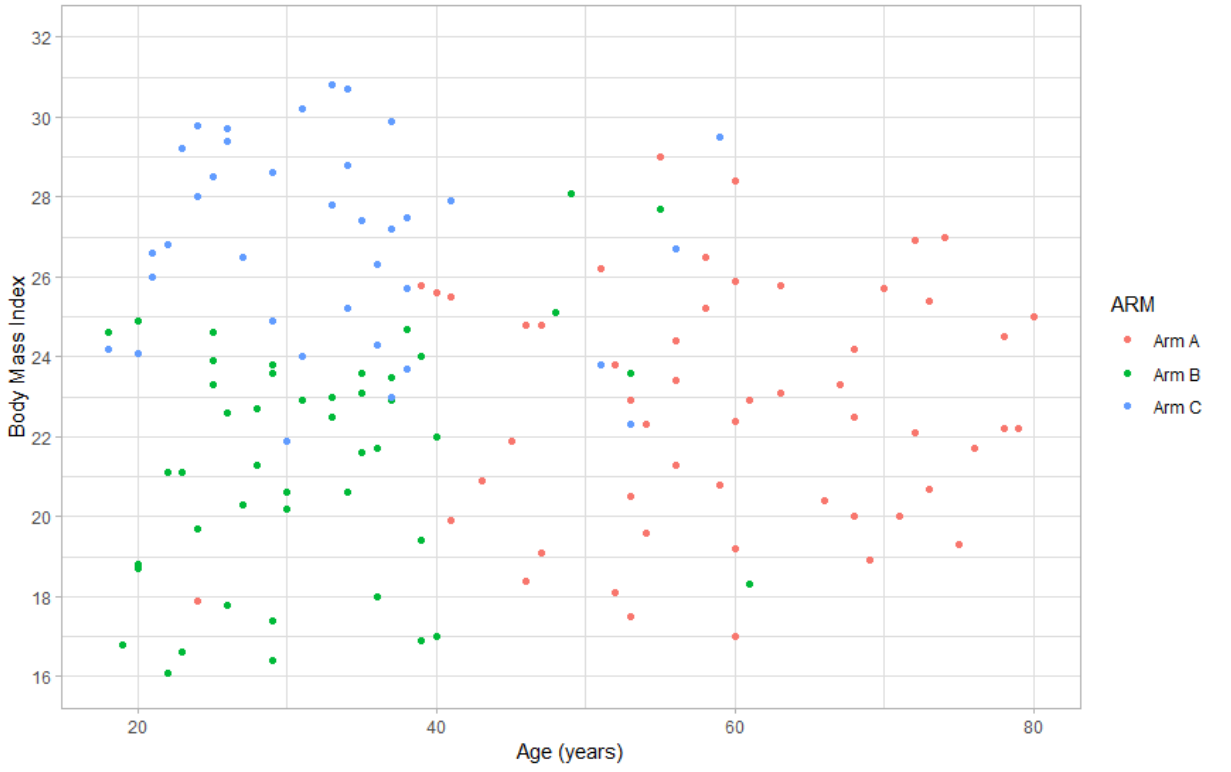



```
ggplot(ASL, aes(x = AGE, y = BMI, col = ARMCD, size = TRTDUR, shape = SEX))+
   geom_point()
```

Figure 6 - Five variables are mapped to different aesthetics.

So, now between x and y dimensions with the color, size and shape aesthetics being used, the plot displays 5 dimensions of the ASL data frame.

We map the data onto different aesthetics, so each aesthetic is a scale. Color or size aesthetics are scales like x or y. In ggplot2 scales determine the way our data is mapped to geom. There are lots of scale functions inside ggplot2 depending on which scale we want to modify and the type of variable that is mapped onto this scale. Every scale function begins with the word "*scale_*". The second part of the function defines which scale we want to readjust. All aesthetics has a corresponding scale function, so we should put the name of the aesthetic as the 2nd part of the function. The third part of the function is a type of data that is used. It should be discrete for categorical variables and continuous for continuous variables. For example, complete scale function looks like that: *scale_x_continuous( )* or *scale_color_discrete( )*. The most common arguments of the scale functions are: name, limits, breaks, expand and label. The first argument is always the name of the scale. Limits describe the scale limits, breaks is an argument that defines brakes of the scale. Expand argument adds some indentation between the data to provide some distance from the axes. In Figure 7 we have used *scale_y_continuous( )* function to change the name of the y scale, to define the limits of this scale and make breaks every two values. Also, we used the *scale_color_discrete( )* function to change the labels in the legend.




```
ggplot(ASL, aes(x=AGE, y =BMI)) +
  geom_point(aes(col = ARMCD)) +
  scale_y_continuous("Body Mass Index", limits = c(16,32),
                     breaks=seq(16,32,2)) +
  scale_color_discrete("ARM", labels = c("Arm A","Arm B","Arm C")) +
  scale_x_continuous("Age (years)")
```

Figure 7 - Influence of scale functions.

1.2 GEOMETRIES

The third essential layer for all plots is a geometry layer. The geometry layer defines which geometry objects will be the visual representation of our data. There are several different geoms in ggplot2 and here I will show the most common ones and how they can influence the plot.

Scatter Plots

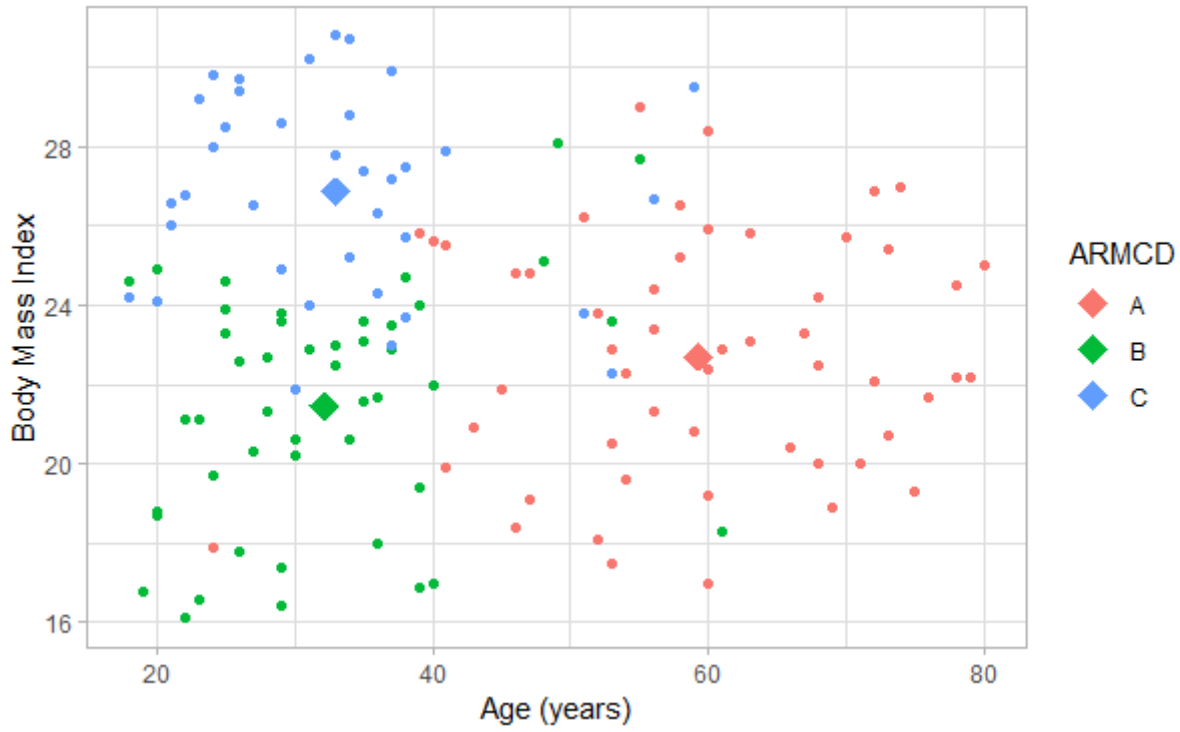
We have already created a few scatter plots before and found out that to create such plots the *geom_point( )* function should be used. A scatter plot creates points on our chart where each point corresponds to an observation in our data. Each geom has specific aesthetic mappings, so they require a different number of variables for mapping or specific types of variables [7]. For instance, to use *geom_point( )* we need to define x and y aesthetic (a type of variable can be discrete and continuous). In addition to essential aesthetics, we can also choose optional aesthetics, as was shown before. For example, for *geom_point( )* we can specify the *color*, *size*, *alpha*, *fill* and *shape* as optional aesthetics, but they also could be attributes.

We previously used only one geom to create a plot, but in ggplot2 we can put many geoms on top of each other. We also figured out that we can specify aesthetics inside the geom functions (see Figure7), but we

also can specify the data layer inside the geoms. This means that it is possible to plot several data sets on top of each other and control the aesthetic mapping of each layer independently. For example, we can use ASL data set and count the mean values of AGE and BMI variables and put these values into ASL_mean data frame:

```
ASL_mean <- ASL %>%
group_by(ARMCD) %>%
summarise(avg_age = mean(AGE),
          avg_bmi = mean(BMI))
```

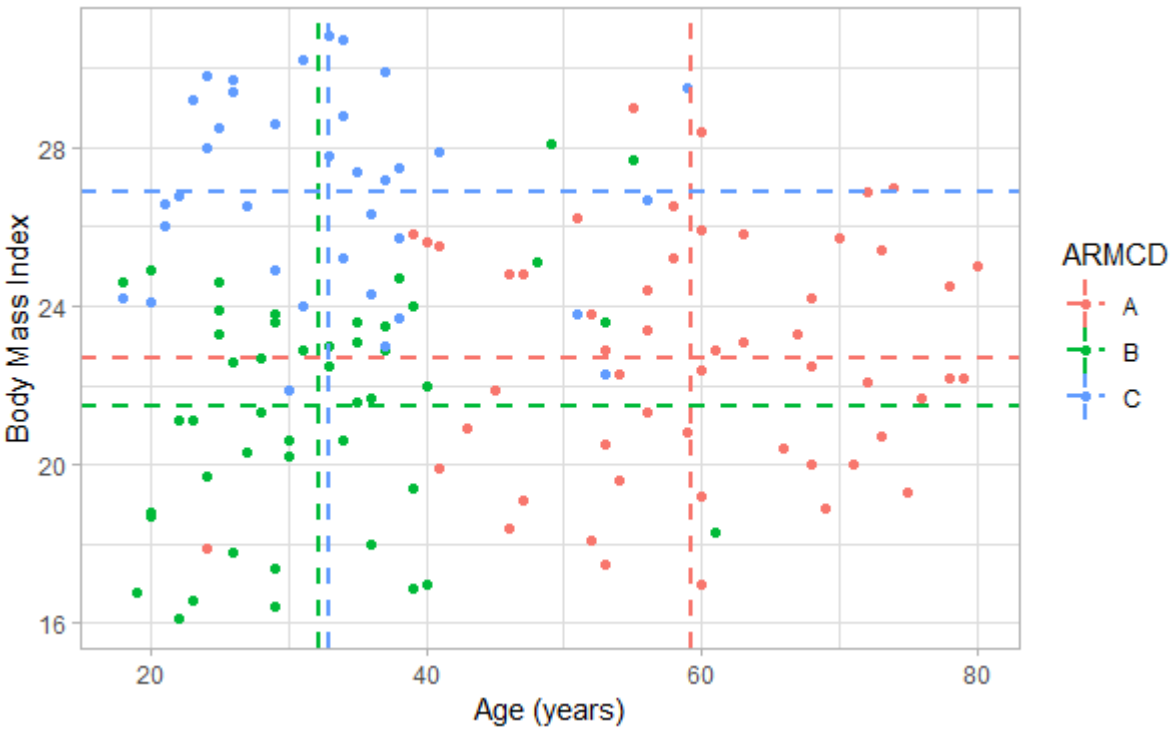
In Figure 8 we used the graph from Figure 2, to which we added the second *geom_point( )* to place the mean values of AGE and BMI from ASL_mean data set on the top of our scatter plot. Now we can see mean values of AGE and BMI for each ARMCD as a big rhombus on our chart. Also, labels can be changes with the help of *labs( )* function.




```
ggplot(ASL, aes(x = AGE, y = BMI, col = ARMCD)) +
   geom_point() +
   geom_point(data = ASL_mean, aes(x = avg_age, y = avg_bmi), shape = 18,
              size = 5) +
   labs(x = "Age (years)", y = "Body Mass Index")
```

Figure 8 - Two different data sets on the same plot.

We can change Figure 8 by plotting the means as intersections of lines. To solve this task we should use two other geoms: *geom_hline( )* which creates horizontal lines and *geom_vline( )* which creates vertical lines. These both geoms require only one x aesthetic, but I have also used the color aesthetic to create lines of the same color as dots (Figure 9). The *linetype* argument was used to create dotted lines (*linetype* also can be an attribute as shown in Table 1.1).

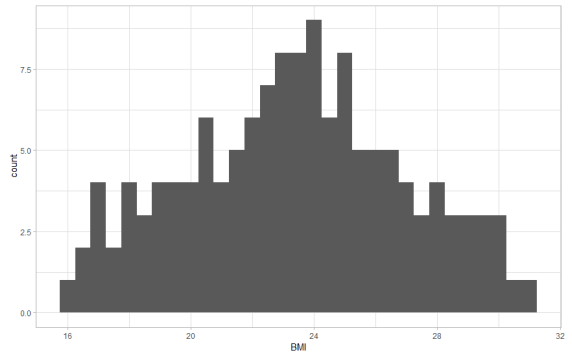



```
ggplot(ASL3, aes(x = AGE, y = BMI, col = ARMCD)) +
  geom_point() +
  geom_vline(data = ASL.summary, aes(xintercept = avg_age, col = ARMCD),
             linetype = 2, size = 1) +
  geom_hline(data = ASL.summary, aes(yintercept = avg_bmi, col = ARMCD),
             linetype = 2, size = 1) +
  labs(x = "Age (years)", y = "Body Mass Index")
```

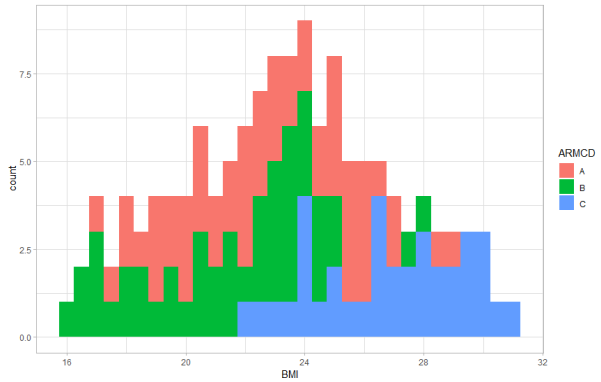
Figure 9 - Three geoms on the one plot.

Bar Plots

One of the most common types of bar plot is a histogram which visualizes the distribution of a single continuous variable by dividing the x-axis into bins and counting the number of observations in each bin. To create a histogram we just need to specify x aesthetic (a continuous variable of interest) and add *geom_histogram( )*. Notice that *geom_histogram( )* plots statistical function to our data set, not the original data. Bins can be adjusted using *binwidth* argument inside the *geom_histogram( )*, as shown in Figure 10a. By default, *stat_bin( )* uses 30 bins and this is not a good default, but the idea is to make you experimenting with various bin widths. As we have already done for a scatter plot, we can color bars according to each value of the categorical variable using the *fill* aesthetic (Figure 10b), but it is not clear if the bars are stacked on each other or overlapping. *geom_histogram( )* has an argument called *position*. The default value of the *position* argument is "stack", so in Figure 10b the bars are stacked on top of each other. The most popular values of the *position* argument except "stack" are "dodge" and "fill".



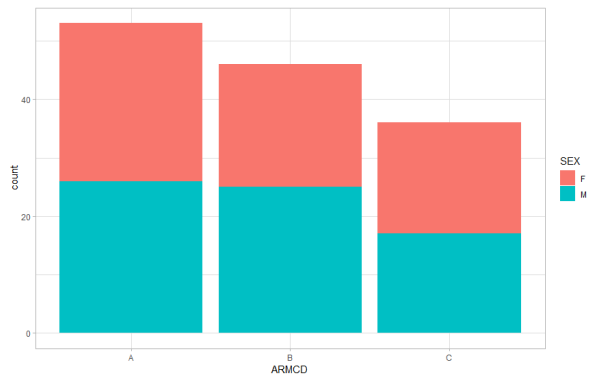

```
ggplot(ASL, aes(x = BMI)) +
  geom_histogram(binwidth = 0.5)
```



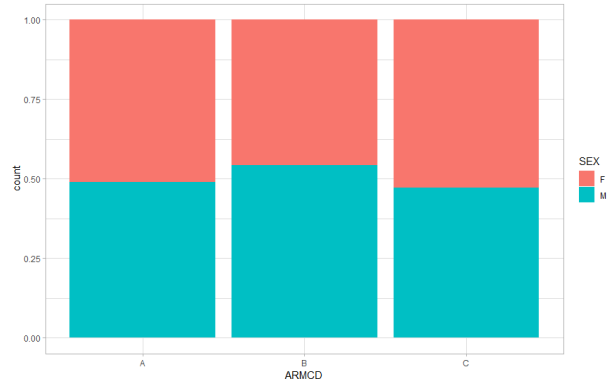

```
ggplot(ASL, aes(x = BMI,
       fill = ARMCD)) +
  geom_histogram(binwidth = 0.5,
position = "stack")
```

Figure 10 - Histogram plots.

The other way to create a bar plot is using the *geom_bar( )* function. The difference between *geom_bar( )* and *geom_histogram( )* is that they use different statistical functions or stat layers. *geom_histogram( )* uses *stat_bin( )* function which can be applied to a continuous variable while *geom_bar( )* uses the *stat_count( )* function which counts the number of times each value of categorical or continuous variable occurs. So, we can use the *geom_bar( )* to create a bar plot for a categorical variable. *geom_bar( )* also has a *position* argument like the *geom_histogram( )* and the default value is also equal to "stack". Here we also have alternative positions like "dodge" or "fill". If the *position* is set to "fill" (Figure 11b) then ggplot2 normalizes each bin to represent the proportion of all the observations in each bin by each category. "Dodge" position splits each data point in a given category as shown in Figure 12a. Also, we set the *position* argument to the *position_dodge( )* function which allows adjusting a horizontal position. For instance, we can overlap the bars using the *width* argument of the *position_dodge( )* function as shown in Figure 12b.

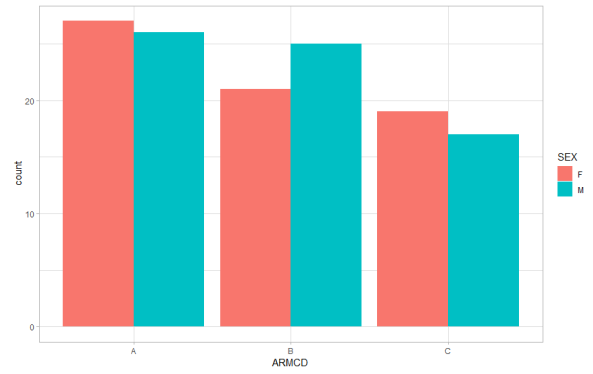



```
ggplot(ASL, aes(x=ARMCD, fill=SEX))+
   geom_bar(position = "stack")
```

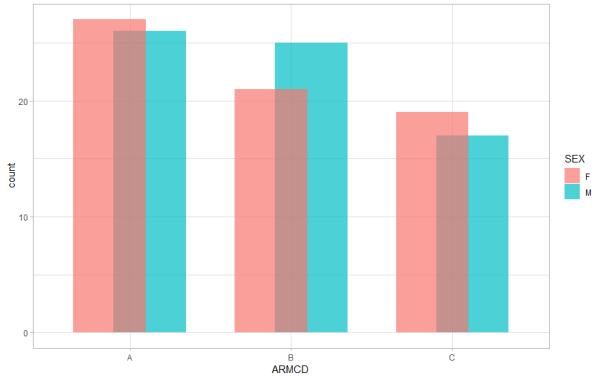



```
ggplot(ASL, aes(x=ARMCD, fill=SEX)) +
  geom_bar(position = "fill")
```

Figure 11 - Bar plots with position argument equal to "stack" and "fill".




```
ggplot(ASL, aes(x=ARMCD, fill=SEX))+
  geom_bar(position = "dodge")
```

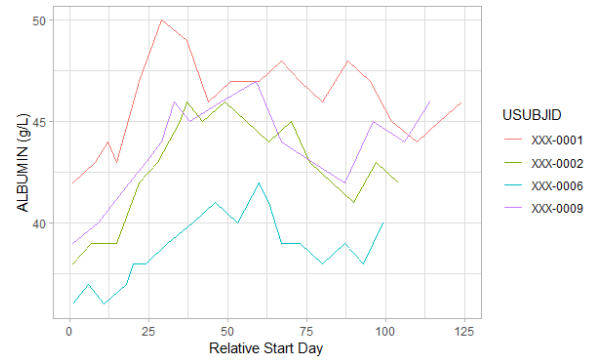



```
ggplot(ASL, aes(x=ARMCD, fill=SEX)) +
  geom_bar(alpha = 0.7, position =
  position_dodge(width = 0.5))
```

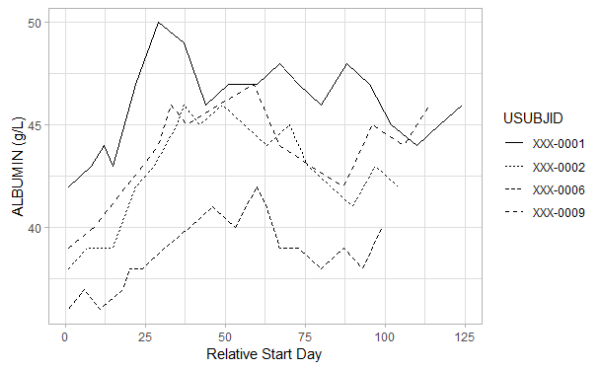
Figure 12 - Bar plots with different types of dodging.

Line Plots

To draw a line plot we simply have to call the *geom_line( )* function and map two variables on x and y aesthetics. If we have a categorical variable and need to draw several distinguishable lines depending on this categorical variable then it is possible to use optional aesthetics like *color* or *linetype*. In Figure 13 we can see how the ALBUMIN parameter taken from ALB data set varies over time for four different patients.




```
ggplot(ALB, aes(x=ADY, y=AVAL, col =
USUBJID)) +
   geom_line() +
   labs(x = "Relative Start Day",
        y = "ALBUMIN (g/L)")
```




```
ggplot(ALB, aes(x=ADY,y=AVAL,
linetype = USUBJID)) +
   geom_line() +
   labs(x = "Relative Start Day",
        y = "ALBUMIN (g/L)")
```

Figure 13 - Line plots with different aesthetic mapping of USUBJID variable.

In this part, I have focused on 3 essential layers – the data, aesthetics, and geometries. As we can see 3 essential layers allow us to create simple graphs quickly. We will explore the remaining layers in the 2nd part and see how they can make our plot more complicated and effective.

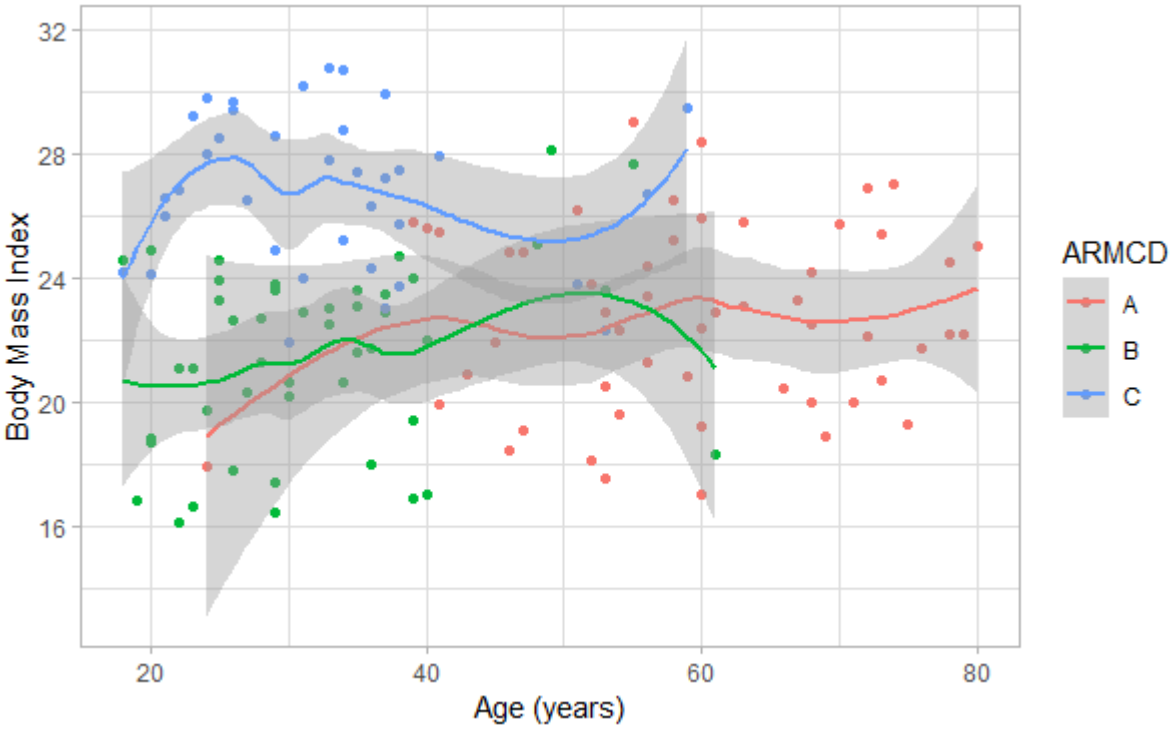
OPTIONAL LAYERS

2.1 STATISTICS

The statistical layer always goes together with the geoms layer, because every geom has a default stat function inside. The difference between the geom and the stat layers is that geoms stand for geometrical objects that appear to be the core elements seen on the plot such as points, lines, bars while stats stand for statistical transformation summarizing the data in various ways, for example, counting observations, adding loess lines (loess short for local regression) or adding a confidence interval to the loess line [8]. Every geom undergoes statistical transformation prior to being plotted like we could see earlier when the *stat_bin( )* was transforming the data to create a histogram and the *stat_count( )* was transforming the data to create a bar chart. So, if we look at Figure 9 and instead of geom_histogram( ) write *stat_bin( )* then we will get absolutely the same result. Geoms are required objects or the "core" of the plot and stats are not required to produce a plot, but can enhance it significantly. So, the statistical layer is always called within the geom function or can be called independently.

stat_bin( ) function which we have already discussed is a default stat for *geom_histogram( )* and *geom_freqpoly( )*. It visualizes the distribution of a single continuous variable by dividing the x-axis into bins and counting the number of observations in each bin. The difference between these two geoms lies in the fact that histograms display the counts with bars and frequency polygons display the counts with lines.

stat_count( ) counts the number of cases at each x position for discrete variables or can be used similarly to *stat_bin( )* for continuous variables. *stat_identity( )* leaves the data unchanged and can be applied to almost all geoms. The simple example of *stat_identity( )* is *geom_point( )* which creates dots depending on the x and y values or *geom_line( )* which simply connect these dots to create lines.




```
ggplot(ASL, aes(x = AGE, y = BMI,  col = ARMCD)) +
  geom_point() +
  geom_smooth() + labs(x = "Age (years)", y = "Body Mass Index")
```

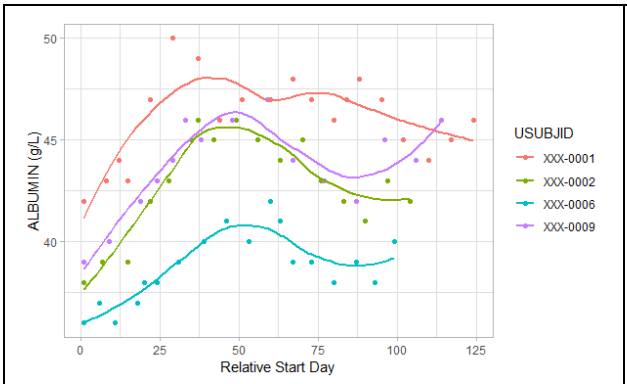
Figure 14 - Loess lines produced by the geom_smooth().

Let's have a closer look at quite commonly used statistics called *stat_smooth( )* that also can be accessed with the *geom_smooth( )* function [9]. Sometimes it is quite difficult to see the trends with only points on the graph, so adding a smoothing line allows us to see what kind of trends are in our data. So, we can use the scatter plot from Figure 2 and add smoothing lines on top of *geom_point( )*. We can see in Figure 14 that *geom_smooth( )* has added smoothing lines for each ARMCD and also confidence intervals on the smooth as well (CI can be removed by setting *se* argument to "FALSE"). You can learn more about the methods of smoothing and counting of the confidence intervals on tidyverse official web page [10].

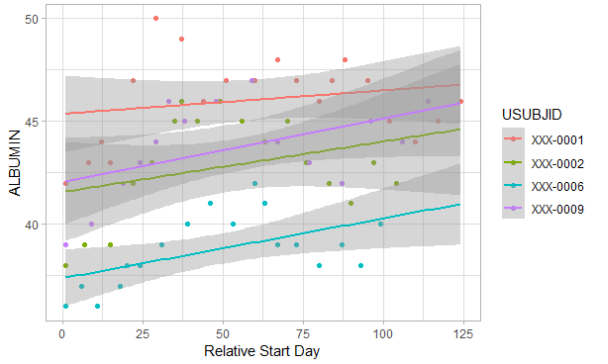
When we call *geom_smooth( )* or *stat_smooth( )* we receive the following message:

```
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

As we see *geom_smooth( )* used the "loess" method which is used for less than 1000 observations, but it can be changed with the help of the *method* argument. For groups larger than 1000 the default *method* is "glm". *Span* argument controls the amount of smoothing for the default loess smoother i.e the degree of smoothing. Smaller spans produce more curly lines and larger spans create smoother lines. In Figure 15 we used ALB data set to show how the specific analysis parameter changes over time for different subjects. Also, we can see that changing of the span argument have made our lines more waved and also the confidence interval has disappeared due to setting *se* argument to "FALSE". In Figure 15b we have set the *method* argument to "lm", so we get a linear model which is not predictive by default meaning that the lines produced by the *stat_smooth( )* are bounded by the limits of the data. But we can change it by setting *fullrange* argument to "TRUE", so the lines are extended to the border of the plot and as expected – the standard error grows with the increase in the distance from the data where we trying to use the estimation. In Figure 16 we have added the general smoothing line for all subjects and have changed the type of line to highlight it.



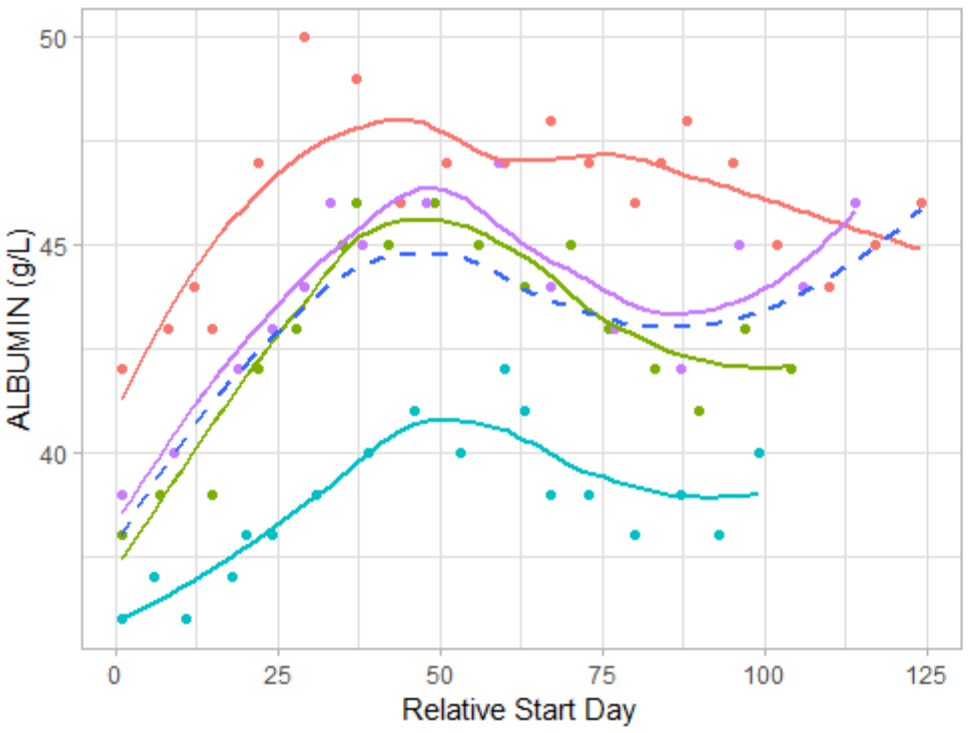

```
ggplot(ALB, aes(x=ADY, y=AVAL,
       col=USUBJID)) +
 geom_point() +
 geom_smooth(se = FALSE, span=0.7)+
 labs(x = "Relative Start Day",
      y = "ALBUMIN (g/L)")
```




```
ggplot(ALB, aes(x=ADY, y=AVAL,
       col=USUBJID)) +
  geom_point() +
  geom_smooth(method = "lm",
              fullrange = TRUE) +
  labs(x = "Relative Start Day",
       y = "ALBUMIN")
```

Figure 15 - Different geom_smooth() settings.

Another quite commonly used statistics function is *stat_summary( )* that summarizes y values at each unique x value. For instance, we need to count the mean value of BMI for each ARMCD. To do it we just need to call the *stat_summary( )* function and specify two arguments: to define what statistics we need to count we should set the *fun.y* argument to "mean" and to define the geometrical object that will show this statistics we should define the *geom* argument. In Figure 17a we set the *geom* argument to "bar" and obtained the bar chart where the height of each bar equals the mean BMI for each specific ARMCD.

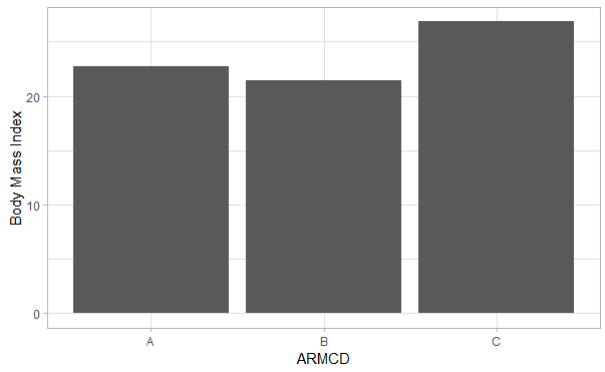



```
ggplot(ALB, aes(x = ADY, y = AVAL, col = USUBJID)) +
  geom_point() +
  stat_smooth(se = FALSE) +
  stat_smooth(aes(group = 1), se = FALSE, linetype = 2) +
  labs(x = "Relative Start Day", y = "ALBUMIN (g/L)")
```

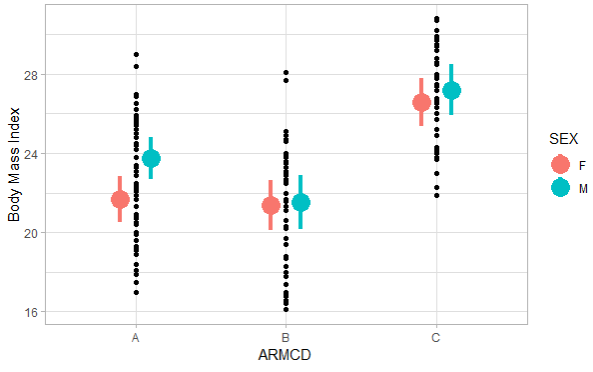
Figure 16 - Adding loess lines for each USUBJID and the overall loess line.

Using the *stat_summary( )* we can count a variety of statistics and show it on the plot with the help of various geoms. Another example is shown in Figure 17b where we display all the BMI values for each ARMCD and the counted mean and CI for BMI inside each ARMCD and SEX. Here we used the *fun.data* argument which takes a function, applies it to the data and returns the data frame with variables: ymin, y, and ymax. We set *fun.data* to “mean_cl_normal” which computes 3 summary variables: the sample mean and the lower and upper Gaussian confidence limits based on the t-distribution. Note that “mean_cl_normal” is the function from Hmisc package that contains a large number of functions for data analysis, high-level graphics, utility operations, functions for computing the sample size and power, importing and annotating data sets, imputing missing values, advanced table making, etc [11]. We have specified the *geom* argument to “pointrange” and have used *position_dodge()* function to separate the pointranges in different directions (otherwise they will overlap the data and themselves). To count separately mean and CI for each SEX inside each ARMCD we have used the *aes( )* argument inside *stat_summary( )* function. Moreover, you can see that in this way we control each layer independently, so it is possible to color pointranges without coloring all the dots.

In this part we have looked a little at the statistical layer, have found out how it is interconnected with the geometry layer and how they differ. We have shown some examples of stat functions and how they can affect the plot. Of course, there are many other statistical functions, but the format of the article does not imply focusing on other ones.




```
ggplot(ASL, aes(x=ARMCD, y=BMI)) +
  stat_summary(fun.y = mean,
               geom = "bar") +
  labs(y = "Body Mass Index")
```




```
ggplot(ASL, aes(x=ARMCD, y=BMI)) +
  geom_point() +
  stat_summary(aes(col = SEX),
    fun.data = "mean_cl_normal",
    geom = "pointrange", size = 2,
position=position_dodge(width=0.4))+
  labs(y = "Body Mass Index")
```

Figure 17 - The stat_summary() capabilities.

2.2 COORDINATES AND FACETS

The coordinates layer defines the coordinate system of a plot and along with the scales controls the location of geoms. The default function in ggplot2 is the *coord_cartesian( )* function that controls XY Cartesian plain of the plot. Also, some properties of this layer can overlap with other functions in ggplot2, but behave very differently. For example, zooming in the plot can be done in several ways:

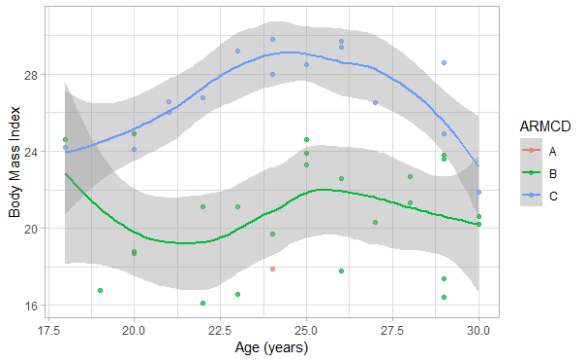
1) We can use the limits argument as we did in Figure 6 inside the scale functions.
2) We can add *xlim( )* or *ylim( )* function directly to our plot.
3) We can set *xlim* or *ylim* arguments inside *coord_cartesian()* function.

For instance, we want to zoom a part of the x-axis in Figure 14. We can use the *scale_x_continuous( )* function as shown in Figure 18a. Acting this way we will receive warning messages that ggplot2 removes the rows containing missing values.

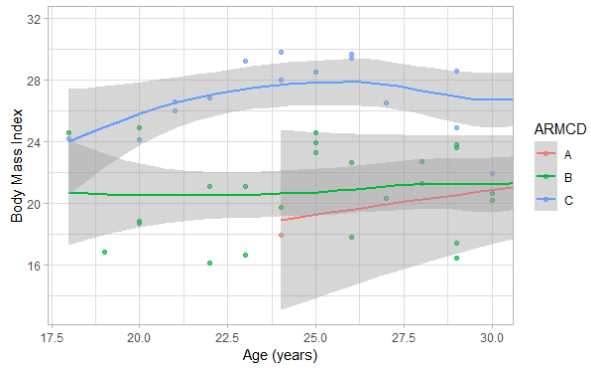
```
Warning messages:
1: Removed 95 rows containing non-finite values (stat_smooth).
2: Removed 95 rows containing missing values (geom_point).
```

It doesn't occur because of missing values in our data set. If we set the *limits* argument inside the scale function to a smaller range than the total range of the data, then the *scale_x_continuous( )* filters our original data set. Also, we can see that red curve produced by *geom_smooth( )* has disappeared since there is only one value for this ARMCD inside these limits, so the loess curve hasn't been calculated. Moreover, we can see that loess smoothing doesn't go beyond the range of the points plotted, but of course, we know that there is some data beyond these limits. The second method using the *xlim( )* function will directly create the same result as well as the first way. The only difference is that we can't specify other arguments that we discussed in Figure 6. The third way is using the *coord_cartesian( )* function and the *xlim* argument inside it, which simply zooms in the plot from Figure 14. In Figure 18b we can see the red curve on the plot despite the fact that there is only one dot in this range for this ARMCD. And we see that the loess curves continue to pass the limits that we have set, which indicates that there

are some data points beyond the plotted range, so *coord_cartesian()* function has literally zoomed the plot from Figure 14. These examples show that the change of limits should always be done carefully to avoid unexpected results.




```
ggplot(ASL, aes(x = AGE, y= BMI,
       col = ARMCD)) +
  geom_point(alpha=0.7) +
  geom_smooth()+
scale_x_continuous(limits=c(18,30))+
  labs(x = "Age (years)",
       y = "Body Mass Index")
```




```
ggplot(ASL, aes(x = AGE, y= BMI,
       col = ARMCD)) +
  geom_point(alpha=0.7)+
  geom_smooth()+
  coord_cartesian(xlim = c(18,30)) +
  labs(x = "Age (years)",
       y = "Body Mass Index")
```

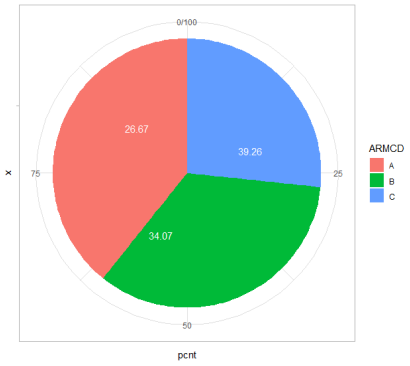
Figure 18 - Two ways to change the limits of the plot.

Another common goal is changing the height-to-width aspect ratio and this can also be done using the coordinates layer. So, to change the aspect ratio we should use the *coord_fixed()* function. The default *ratio* = "1" (expressed as y / x), so one unit on the x-axis is of the same length as one unit on y-axis. To change the aspect we should use the *ratio* argument in the *coord_fixed(ratio =)* function. A ratio higher than one makes units on the y-axis longer than units on the x-axis, and vice versa [12]. But there are some exceptions to this rule, for instance when ranges differ significantly. The *coord_equal()* function is a specific case of the *coord_fixed( )* function and it is equal to *coord_fixed(ratio = 1)*. The *coord_flip( )* function swaps x and y aesthetics, so x becomes vertical from bottom to top and y – horizontal from left to right.

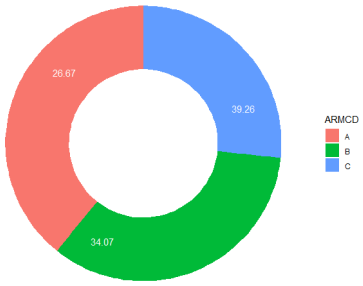
To produce a pie chart we should use the *coord_polar( )* function which maps x or y scale to the angle (*theta* argument). For instance, we need to show the proportion of patients between arms. Firstly, we should prepare the data to create a pie chart: we need to count percent values manually and create *y_pos* variable which contains the position of these percent values. It can be done using the following code [13]:

```
ASL.prop <- ASL3 %>%
  group_by(ARMCD) %>%
  tally()
ASL.prop <- ASL.prop %>%
  mutate(tot_subj = sum(n),
         pcnt = round(100*(n/tot_subj),digits = 2))
ASL.prop <- ASL.prop %>%
  arrange(desc(pcnt)) %>%
  mutate(y_pos = cumsum(pcnt)-0.5*pcnt)
```

To create a pie chart we simply call the *geom_bar()* with the *stat* argument set to "identity", add a coordinates layer by calling *coord_polar()* and add *geom_text()* to show the percent values on the pie chart (see Figure 19). To create a donut chart we simply can add the *xlim()* function that will make a hole inside the pie chart. Also, we added *theme_void()* which is useful for a plot with non-standard coordinates like in our example.




```
ggplot(ASL.prop, aes(x = "", y=pcnt,
       fill = ARMCD)) +
 geom_bar(stat = "identity") +
 coord_polar(theta = "y") +
 geom_text(aes(y=y_pos, label=pcnt),
           color = "white")
```

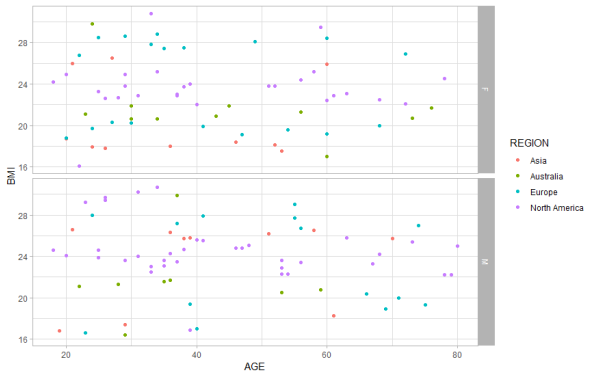



```
ggplot(ASL.prop, aes(x = 2, y = pcnt
,
       fill = ARMCD)) +
 geom_bar(stat = "identity") +
 coord_polar(theta = "y") +
 geom_text(aes(y=y_pos, label=pcnt),
           color = "white") +
 xlim(0.5,2.5) + theme_void()
```

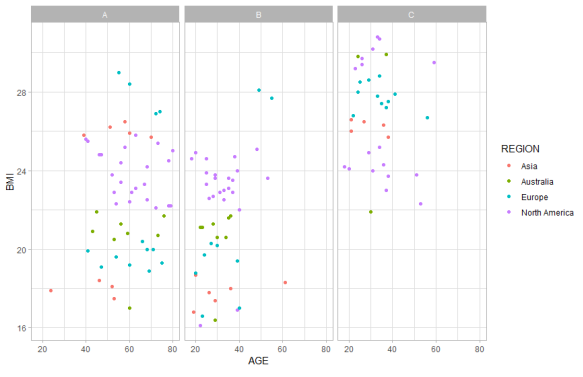
Figure 19 - Pie chart and donut chart.

Facets

The basic idea of facets is that we can divide one large complex graph into several small ones with exactly the same coordinate system. For example, we can use the graph from Figure 1 and at this point it is not necessary to color the dots according to the values of ARMCD variable, but each ARMCD has to be on a different plot. To achieve it we can add facets layer by calling the *facet_grid( )* function. As an argument *facet_grid( )* takes two variables which are separated by a tilde. The variable specified on the left splits up the plot into rows and the variable on the right splits it up into columns: *facet_grid(rows ~ columns)*. If we don't need to split up the plot into both rows and columns then we put a dot instead of the variable name as shown in Figure 20a where we have divided the plot into 3 columns depending on the ARMCD. In Figure 20b we have used SEX variable to split up the plot into rows. Now on each plot we show a different subset of the data, so it is easy to compare them.



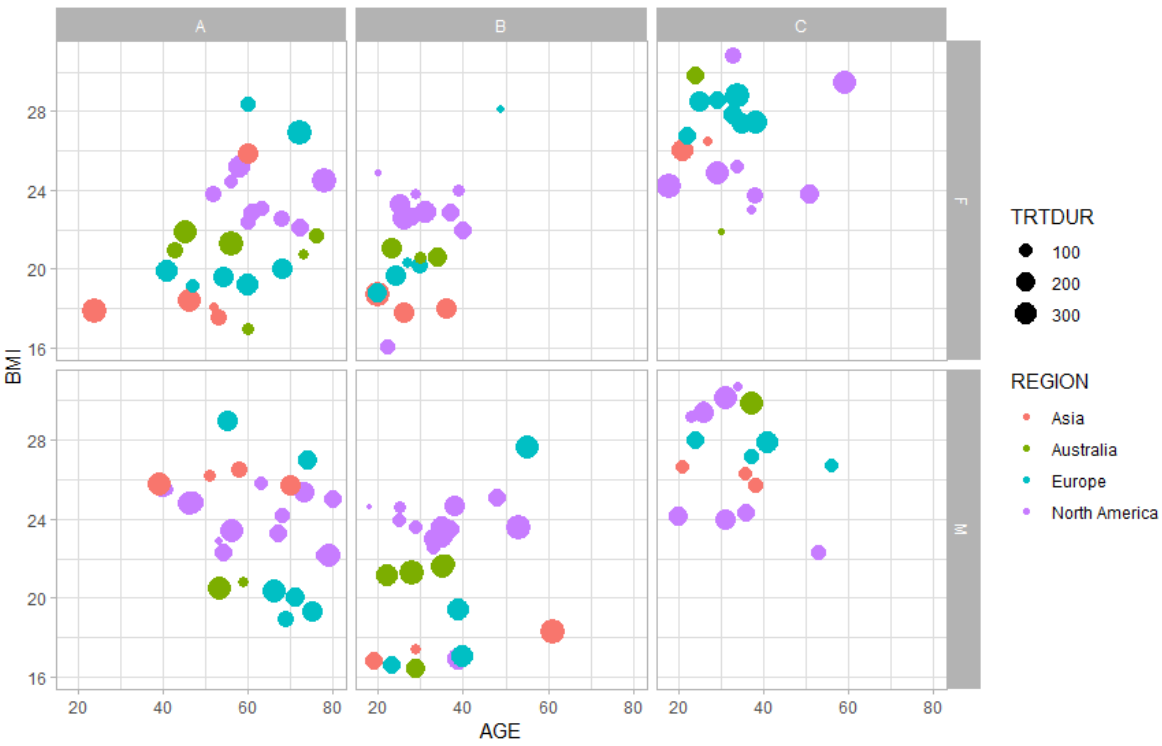

```
ggplot(ASL, aes(x = AGE,y = BMI,
       col = REGION)) +
  geom_point() + facet_grid(SEX ~ .)
```




```
ggplot(ASL, aes(x = AGE,y = BMI,
       col = REGION ))+
 geom_point() + facet_grid(. ~ ARMCD)
```

Figure 20 - Splitting the plot into rows and columns.

In Figure 21 we have split up the plot into rows and columns using SEX and ARMCD variables accordingly. Moreover, we have set TRTDUR to the *size* aesthetic and REGION to the *color* aesthetic. So, on this plot we can see six subsets of our data, which can be easily compared. If there is a large number of values in a categorical variable, then we can control the number of columns using the *ncol* argument or we can allow the scales to vary by setting the *scales* argument to "free" [14].




```
ggplot(ASL, aes(x = AGE, y = BMI, col = REGION, size = TRTDUR)) +
  geom_point() +
  facet_grid(SEX ~ ARMCD)
```

Figure 21 - Splitting plot into rows and columns simultaneously.

2.3 THEMES

The themes layer controls all the visual elements of the plot that are not part of the data. So, themes don't add words or change ranges of variables, they change the font, size, color, etc. Visual elements are divided into 3 different groups: text, line, and rectangle. To change each type of elements we need to call an appropriate function inside the *theme( )* function. There are 4 basic functions inside the themes layer. Using the *element_text( )* function we can modify all text elements of the plot. The *element_line( )* function changes lines, the *elements_rect( )* modifies borders and background and the *element_blank( )* function just removes all the items of the plot, so nothing will be drawn at all. The theme layer also allows us to modify positions of elements.

Each text element has its own unique name, so we have two ways of changing the text on the plot: we can access all the text elements together, but also each specific element such as the text on the axis or the title of the legend or plot separately.

Lines include axis lines, tick marks on the axis and all grid lines (major and minor). The remaining visual elements are all rectangles of different sizes. We can access rectangles together or each specific rectangle separately in the same way as it was shown for line and text elements. For instance, we can simply alter the background of legend.

So, there is a high number of non-data elements that can be modified by the theme layer and lots of arguments that control those visual elements. You can read more about modifying themes on the official web page of tidyverse [15] or on datanovia blog about ggplot2 [16].

Also, there are many built-in theme templates that can be used in ggplot2 or you can install the ggthemes package that contains a variety of them. The default theme in ggplot2 is the *theme_gray( )* with gray background color and white grid lines, but for this article I changed the default theme to theme_light( ). Here are examples of several themes that exist in ggplot2:

- *theme_bw( )*: white background and gray grid lines
- *theme_light( )*: light gray lines and axis
- *theme_linedraw( )*: a theme with black lines on the white background
- *theme_classic( )*: similar to the *theme_linedraw( )*, but with light grey lines and axes and without grid lines
- *theme_void( )*: an empty theme that can be useful for plots with non-standard coordinates (example of the *theme_void( )* is shown in Figure 19).

If we need to use one specific theme we can set it as default at the beginning of work using the *theme_set( )* function. So, all plots will automatically have the same theme. Also, we can return to the default theme using the *theme_set(original)* command. And, moreover, we can alter any settings of these themes using the *themes_update( )* function.

CONCLUSION

In conclusion, I will show how to combine all these layers that were considered before to create a swimlane plot (see Figure 22) which is quite commonly used in oncology studies. To create this plot we have used transformed ARS data set which contains the following variables: USUBJIDR (a text variable which contains a unique subject identifier that is merged with the best confirmed overall response), ARMCD, DURM (duration in month), FSPD (the first progression disease), FSTCRPR (first CR/PR), DEATH (day of death), LDD (last dosing date). To create a bar chart the *geom_bar( )* function was used and the *coord_flip( )* function to turn it around. As the first PD, the first CR/PR, the relative day of death and the last dosing date are individual variables, they were added as separate layers with help of *geom_point( )* functions. Basically, at this moment the plot is ready and the rest of the code only changes the appearance of the graph. The *labs( )* function adds title, footnote and deletes the label of the y-axis. The *scale_y_continuous( )* function re-defines the label of x-axis and determines the breaks, the *scale_fill_discrete( )* alters the legend for the ARMCD variable. To create a common legend for milestones which unites four *geom_point( )* functions we have used a specific trick: you can see that in every call of *geom_point( )* function we set the color aesthetic to the name of milestone (not to the names of colors or variables). It is done to get a common color scale for these four geoms, which can be accessed using the *scale_color_manual( )* function where we also defined the title of the legend and colors of the dots (the *color* aesthetic here defines the color of dots' frame, not the color of the dot itself). By using the *guides( )* function we have added the legend for milestones and have defined the shapes and colors (using *fill* argument) of the figures shown in the legend. So, we have defined our milestone figures two times – inside the *geom_point( )* functions we set their view in the graph and inside the *guides( )* function - their appearance in the legend. The *theme_bw( )* function is a standard ggplot2 theme which creates a white background and grey grid lines. By using the *theme( )* function we removed vertical grey grid lines, set the position of legends and encircled them with grey rectangles.

As a corollary, it is quite easy to start using ggplot2 in your analysis, the naming convention of ggplot2 allows to easily understand the meaning of a function and what it is responsible for. The idea of layers is very convenient and powerful and becomes clear almost from the very beginning of using ggplot2. Also, there is a great ggplot2 manual with examples and a lot of various material on the web that will provide assistance properly and promptly.

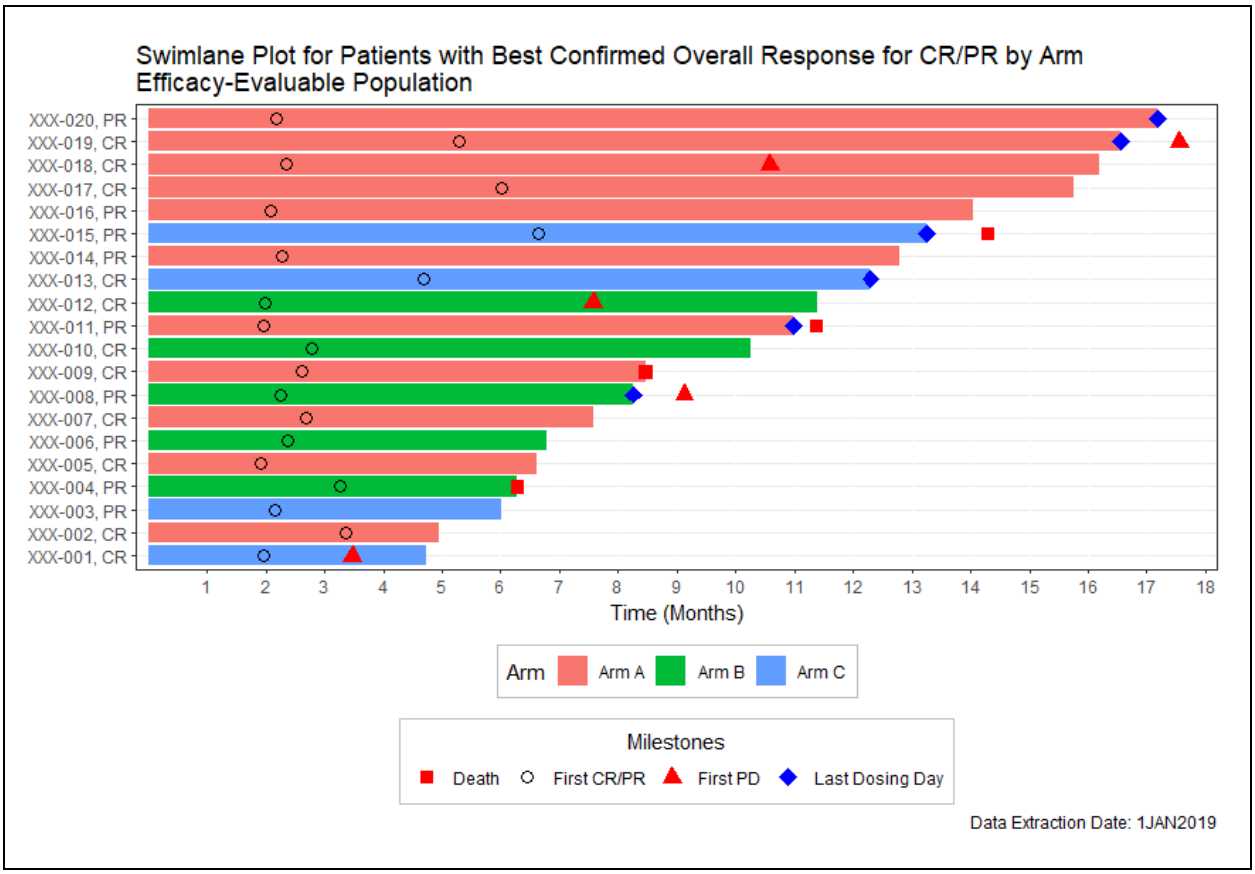


Figure 22 - Swimlane plot.

The code which created the swimlane plot in Figure 22:

```
ggplot(ARS, aes(x = USUBJIDR, y = DURM, fill = ARMCD)) +
  geom_bar(stat = "identity") +
  geom_point(aes(y = FSPD, col = "First PD"), shape=24, fill="red", size=3) +
  geom_point(aes(y = FSTCRPR, col="First CR/PR"), shape = 1, size = 3) +
  geom_point(aes(y = DEATH, col="Death"), shape = 22, fill = "red", size=3) +
  geom_point(aes(y = LDD,col="Last Dosing Day"),shape=23,size=3,fill="blue")+
  coord_flip() +
  labs(x = NULL, title = "Swimlane Plot for Patients with Best Confirmed Over
all Response for CR/PR by Arm \nEfficacy-Evaluable Population",
       caption = "Data Extraction Date: 1JAN2019") +
  scale_y_continuous("Time (Months)", limits = c(0,18), breaks=seq(1,22,1),
                     expand = c(0,0.2)) +
  scale_fill_discrete("Arm", labels = c("Arm A","Arm B","Arm C")) +
  scale_color_manual("Milestones", values = c("First PD" = "red",
                     "First CR/PR" = "black", "Death" = "red",
                     "Last Dosing Day" = "blue"), guide = "legend") +
  guides(col = guide_legend(override.aes = list(shape = c(22,1,24,23),
               fill = c("red","black","red","blue")), order = 2,
               title.position = "top", title.hjust = c(0.5)),
         fill = guide_legend(override.aes = list(shape = NA), order = 1)) +
  theme_bw() +
  theme(legend.position = "bottom", legend.box = "vertical",
        panel.grid.major.x = element_line(color = "white"),
        panel.grid.minor.x = element_line(color = "white"),
        legend.background = element_rect(linetype = "solid", color = "grey"))
```

REFERENCES

1. Joseph Rickert. “What is tidyverse?”. R Views 2017-06-08. Available at https://rviews.rstudio.com/2017/06/08/what-is-the-tidyverse/
2. Overview of ggplot2 available at https://ggplot2.tidyverse.org/
3. Hadley Wickham. “A Layered Grammar of Graphics”. Journal of Computational and Graphical Statistics. Volume 19, 2010. Available at http://vita.had.co.nz/papers/layered-grammar.pdf
4. Garrett Grolemund, Hadley Wickham. “R for Data Science”. O’Reilly, January 2017. Available at https://r4ds.had.co.nz/introduction.html
5. “Plotting with ggplot2” http://monashbioinformaticsplatform.github.io/2015-11-30-intro-r/ggplot.html
6. “Shapes and line types”. Available at http://www.cookbook-r.com/Graphs/Shapes_and_line_types/
7. “Data Visualization with ggplot2” Cheat Sheet by RStudio®. Available at https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf
8. “Loess Regression with R”. Available at http://r-statistics.co/Loess-Regression-With-R.html
9. “Smoothed conditional means” https://ggplot2.tidyverse.org/reference/geom_smooth.html
10. “Summarise y values at unique/binned x”. Available at https://ggplot2.tidyverse.org/reference/stat_summary.html
11. “Hmisc: Harrell Miscellaneous”. Available at https://cran.r-project.org/package=Hmisc
12. “Cartesian coordinates with fixed aspect ratio”. Available at https://ggplot2.tidyverse.org/reference/coord_fixed.html
13. “How to create a pie chart in R using ggplot2”. Available at https://www.datanovia.com/en/blog/how-to-create-a-pie-chart-in-r-using-ggplot2/
14. “Lay out panels in a grid”. Available at https://ggplot2.tidyverse.org/reference/facet_grid.html
15. “Modify components of a theme”. Available at https://ggplot2.tidyverse.org/reference/theme.html
16. “ggplot themes gallery”. Available at https://www.datanovia.com/en/blog/ggplot-themes-gallery/

RECOMMENDED READING

- “Data Visualization with ggplot2” interactive course (in 3 parts) on datacamp.com by Rick Scavetta
- Garrett Grolemund, Hadley Wickham. 2017. *R for Data Science*. Available at https://r4ds.had.co.nz/
- Overview of tidyverse package available at https://www.tidyverse.org
- Andrie de Vries, Joris Meys. 2015. *R For Dummies.* 2nd Edition
- “Data Visualization with ggplot2” Cheat Sheet by RStudio®

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Oleksandr Babych

Experis Clinical Solutions / Intego Group LLC

+1 (407) 512 1006 (Ext. 2449)

oleksandr.babych@intego-group.com

www.intego-group.com