Dapr1: Notes on the live R session, Week 5

In this week's topic, we will see how to define and plot functions. We will also show how to calculate z-scores to standardize a variable. Before we start, let's load "tidyverse":

```
library(tidyverse)
```

Functions

Now, let's consider the following linear function:

$$y = f(x) = 10 + 2x$$

We can create a small dataset that includes a list of "x" input values, along with a list of "y" output values that are the result of applying the function. We will use "tibble()" to make the dataset. The "x" column contains the integers 1-8. The "y" column contains the output of applying the above function to each item in the "x" column. We store the dataset with the name "func_x". Finally, after making the dataset, we can print it out by simply typing its name "func_x":

```
func_x <- tibble(
  x = c(1,2,3,4,5,6,7,8),
  y = 10 + (2*x)
)

func_x
```

```
## # A tibble: 8 x 2
##       x     y
##   <dbl> <dbl>
## 1     1    12
## 2     2    14
## 3     3    16
## 4     4    18
## 5     5    20
## 6     6    22
## 7     7    24
## 8     8    26
```

We can see that each item in the "y" column is the result of applying the function to the corresponding number in the "x" column.

In R you can also define a function and save it with a name, so that you can use it later.

In this case, let's save the above function using the name "fun1":

```
fun1 <- function(x) 10+(2*x)
```

Now that we have defined the function, we can use it to find the output y value for any given x value, or any vector of x values as input:

```
fun1(5)
```

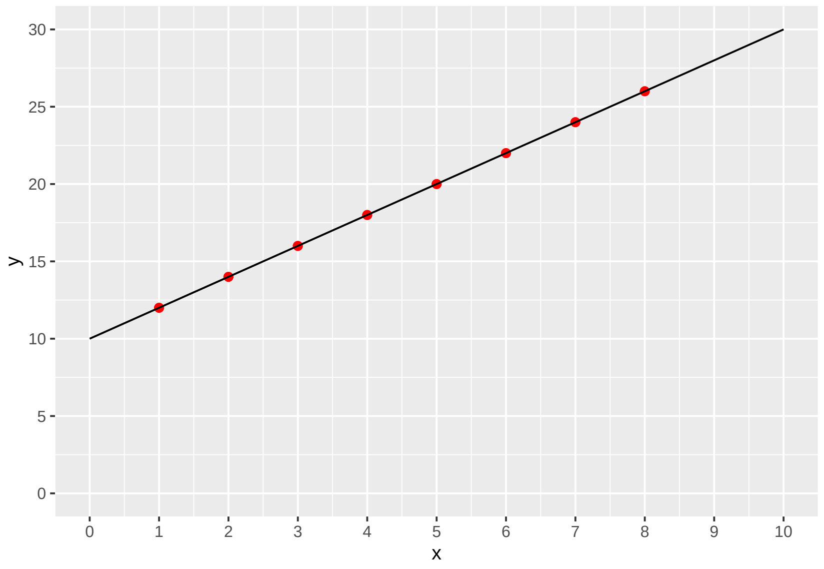
```
## [1] 20
```

```
fun1(c(10,12,14))
```

```
## [1] 30 34 38
```

Let's see how to plot a function. In the following example, we use "geom_point()" to plot a sequence of dots corresponding to the "x" and "y" values of the dataset we created above with the name of "func_x". In addition, we use "stat_function()" to plot the line that corresponds to the same function, which we have defined as "fun1". We also define breaks, labels and limits for the x and y-axes. Note that "c(seq(0,30,5))" evaluates to an ascending list of numbers beginning with 0 and ending with 30, in increments of 5. The term "breaks" is used to specify the major divisions in the plot. The term "labels" is used to specify how these divisions are labelled on the axis. Then, "limits" specifies the maximum and minimum values that will be used on a given axis for the plot.

```
ggplot(func_x, aes(x, y)) +
  geom_point(colour = "red", size = 2) +
  stat_function(fun = fun1) +
  scale_x_continuous(name = "x", breaks = c(0:10),
                     labels = c(0:10), limits = c(0,10)) +
  scale_y_continuous(name = "y", breaks = c(seq(0,30,5)),
                     labels = c(seq(0,30,5)), limits = c(0,30))
```



We can see that the plot produces a straight line, because it is a linear function. In general, a linear function has the following form, where M is the slope and c is the y-intercept:

$$y = Mx + c$$

In our example, the slope is 2. The slope is positive, and this is reflected in a line that ascends from left to right on the graph: each increase of 1 unit on the x-axis corresponds to an increase of 2 units on the y-axis. In our example, the y-intercept is 10: the plotted line crosses the y-axis at a value of $y = 10$.

Non-linear functions can also be plotted in R. Let's consider the following non-linear function:

$$y = f(x) = 5 + x^2$$

First, we will define the function in R, and save it with the name "fun2":

```
fun2 <- function(x) 5+(x^2)
```

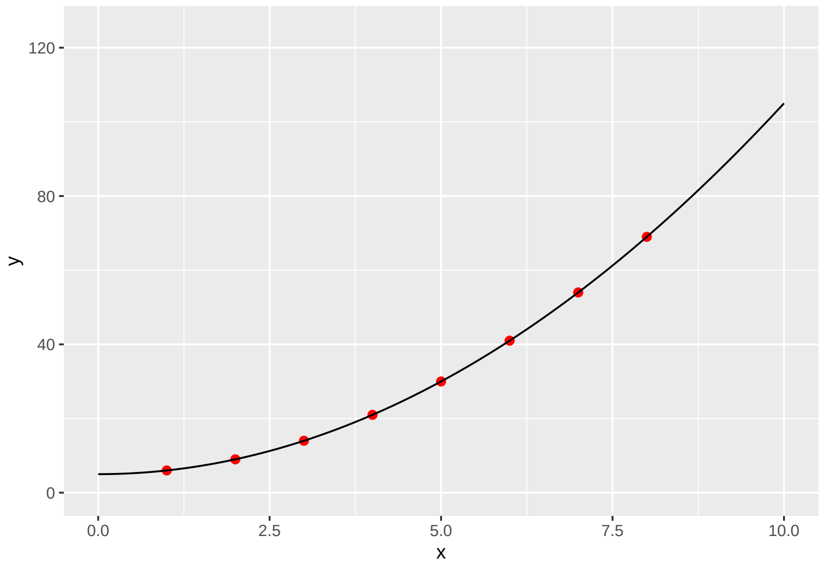
Now, we will make a small dataset with some “x” and “y” values for this function. We will save this dataset with the name “func_x2”, and type the name to show the table, verifying that it is correct.

```
func_x2 <- tibble(
  x = c(1,2,3,4,5,6,7,8),
  y = fun2(x)
)
func_x2
```

```
## # A tibble: 8 x 2
##       x     y
##   <dbl> <dbl>
## 1     1     6
## 2     2     9
## 3     3    14
## 4     4    21
## 5     5    30
## 6     6    41
## 7     7    54
## 8     8    69
```

Finally, we will plot these “x” and “y” values using “geom_point()”, and we will also plot the function as a line, using “stat_function()”. This time, we will allow R to choose the breaks and the labels for each axis, but we will specify the limits:

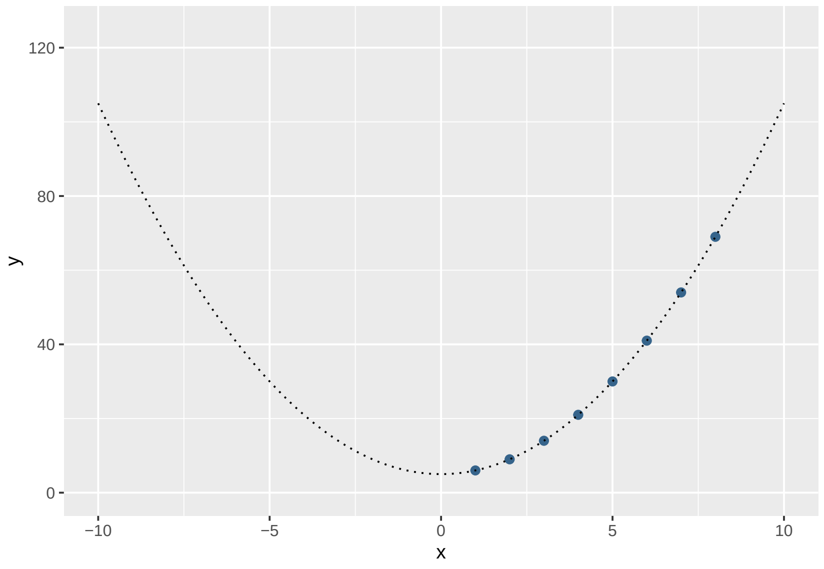
```
ggplot(func_x2, aes(x, y)) +
  geom_point(colour = "red", size = 2) +
  stat_function(fun = fun2) +
  scale_x_continuous(limits=c(0,10)) +
  scale_y_continuous(limits=c(0,125))
```



We can see that the line is curved. This is because the function involves x^2.

Now let’s change a couple of things in the plot. Here, we will use a different colour for the points (colour = “steelblue4”). We also specify that the function should be plotted with a dotted line (linetype=“dotted”). We will also change the limits of the x-axis to show both negative and positive values (limits=c(-10,10)).

```
ggplot(func_x2, aes(x, y)) +
  geom_point(colour = "steelblue4", size = 2) +
  stat_function(fun = fun2, linetype="dotted") +
  scale_x_continuous(limits=c(-10,10)) +
  scale_y_continuous(limits=c(0,125))
```



Standardizing using z-scores

In statistics, z-scores are used to standardize variables. We will explain how this works using a small fictional example. Imagine that I go into my garden and I measure the height of 10 flowers. My tape measure only shows inches, so I take the data and convert them into cm, using the formula $cm = inch * 2.54$. I create a dataset including columns for both cm and inches, using the name "flowers_data":

```
flowers_data<- tibble(inch = c(2,3,1,5.4,6.25,7.02,5.3,15,7.8, 2.5),
                      cm = inch*2.54)
flowers_data
```

```
## # A tibble: 10 x 2
##     inch    cm
##    <dbl> <dbl>
##  1  2     5.08
##  2  3     7.62
##  3  1     2.54
##  4  5.4  13.7
##  5  6.25 15.9
##  6  7.02 17.8
##  7  5.3  13.5
##  8 15    38.1
##  9  7.8  19.8
## 10  2.5   6.35
```

Now let's see what the means and standard deviations are:

```
flowers_data |> summarise(inch_mean = mean(inch),
                           cm_mean = mean(cm),
                           inch_sd = sd(inch),
                           cm_sd = sd(cm))
```

```
## # A tibble: 1 x 4
##   inch_mean cm_mean inch_sd cm_sd
##       <dbl>   <dbl>   <dbl> <dbl>
## 1      5.53    14.0    4.03  10.2
```

Now, let's standardize both the inch data and the cm data. We will do this by converting these measurements into z-scores. Recall the formula for the z-score:

$$z_{x_i} = \frac{x_i - \bar{x}}{s_x}$$

According to the formula, for each individual observation x_i, we subtract the mean $\bar{x}$, and then divide the result by the standard deviation s_x. In other words, the z-score for x_i answers the following question: How many standard deviations above or below the mean is x_i? For example, a z-score of 2 would mean that the data point was 2 standard deviations above the mean, while a z-score of -1.5 would mean that the data point was 1.5 standard deviations below the mean. Let's convert both the cm and the inch data to z scores. Note, in this example, we are using "mutate()" to add two extra columns to the "flowers_data". And, we are saving the result in an updated version of "flowers_data", because we will use the extra columns later. That's why we use "flowers_data <-" at the start of the command, as that will result in the extra columns being kept in the data frame.

```
      flowers_data <- flowers_data |> mutate(cm_z = (cm-mean(cm))/sd(cm),
                                             inch_z = (inch-mean(inch))/sd(inch))
```

If we now type "flowers_data", R-studio will print out a table of the new data frame, including the extra columns with the z-scores (here we pipe it to "round(2)" to tidy up the output)

```
flowers_data |> round(2)
```

```
## # A tibble: 10 x 4
##     inch    cm  cm_z inch_z
##    <dbl> <dbl> <dbl>  <dbl>
##  1  2     5.08 -0.88  -0.88
##  2  3     7.62 -0.63  -0.63
##  3  1     2.54 -1.12  -1.12
##  4  5.4  13.7  -0.03  -0.03
##  5  6.25 15.9   0.18   0.18
##  6  7.02 17.8   0.37   0.37
##  7  5.3  13.5  -0.06  -0.06
##  8 15    38.1   2.35   2.35
##  9  7.8  19.8   0.56   0.56
## 10  2.5   6.35 -0.75  -0.75
```

We can see that all the z-scores for the inch and cm data are the same, even though the original data had been given on different scales. The z-score transformation has mapped both the cm and the inch data onto a common scale. Now, if we check the mean and standard deviation of the z-scores, they should be 0 and 1 (or very close, given rounding error).

```
flowers_data |> summarise(mean_z = round(mean(cm_z),2),
                          sd_z = round(sd(cm_z),2))
```

```
## # A tibble: 1 x 2
##   mean_z  sd_z
##    <dbl> <dbl>
## 1      0     1
```

After a variable has been standardized with a z-transformation, the mean will always be 0 and the standard deviation will always be 1.

Incidentally, we can also calculate z scores for a variable using R's built-in "scale()" function. For example, the following will report the z scores for the cm data:

```r
scale(flowers_data$cm) |> round(2)
```

```
##        [,1]
##  [1,] -0.88
##  [2,] -0.63
##  [3,] -1.12
##  [4,] -0.03
##  [5,]  0.18
##  [6,]  0.37
##  [7,] -0.06
##  [8,]  2.35
##  [9,]  0.56
## [10,] -0.75
## attr(,"scaled:center")
## [1] 14.03858
## attr(,"scaled:scale")
## [1] 10.23751
```

You can verify that the output matches the one for our own calculation.