High Quality Graphics in R

true

January 04, 2021

Contents

Introduction

Welcome to *High Quality Graphics in R*!

In our work, we often use visualizations in two ways: On the one hand, we use them as an exploratory tool to get an overview of our data as quickly and easily as possible. On the other hand, especially in our publications, we want to create highly individualized figures that make a professional impression and convey as quickly as possible the information we want to communicate to the readers of our publications.

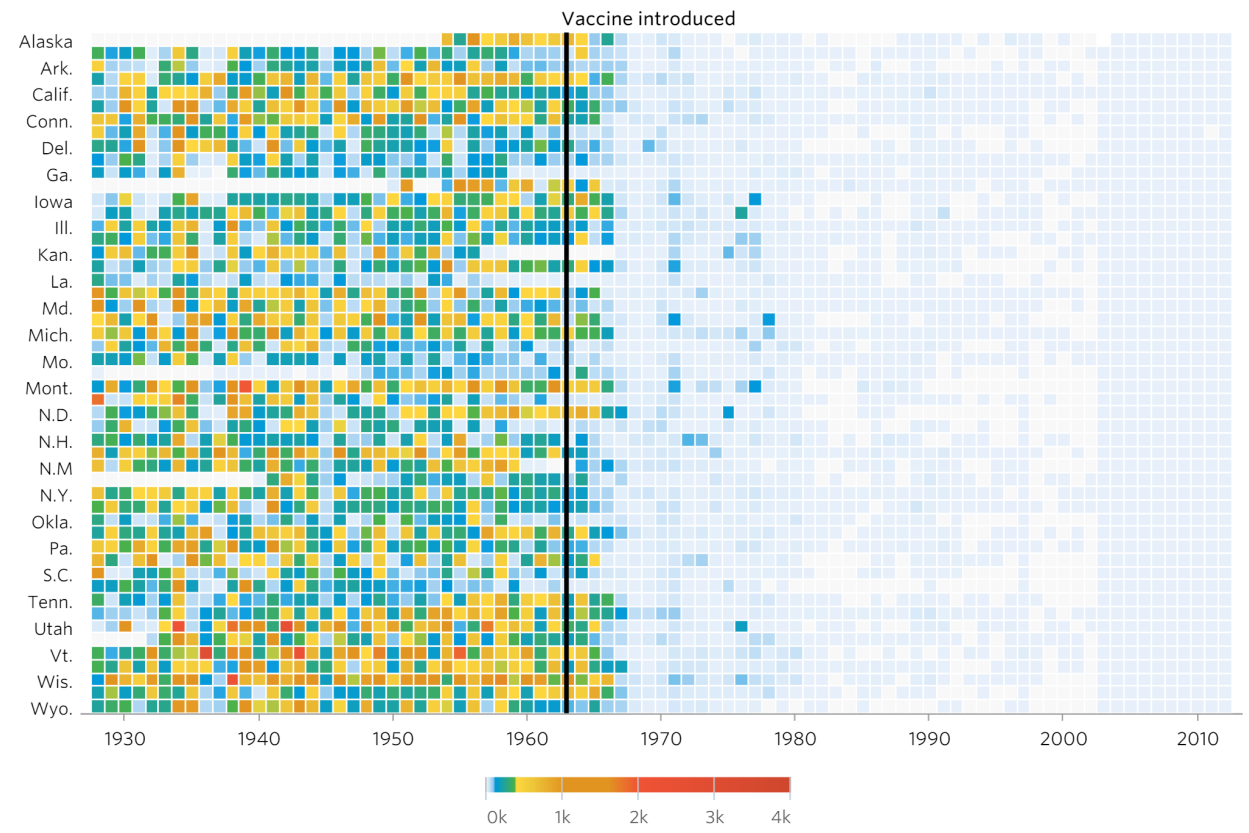


Figure 1: Caption

A particularly impressive example of a good visualization can be found here:

https://www.mikelee.co/posts/2017-06-28-wsj-measles-vaccination-chart/

https://benjaminlmoore.wordpress.com/2015/04/09/recreating-the-vaccination-heatmaps-in-r/

Plotting in base R

When installing base R, some dummy/training data sets are made available which we can easily use by loading the data into memory.

In this example, we use the `DNase` data set which is, roughly speaking, a quantification of the activity of DNase.

Let´s have a look at the first couple of rows of our data.

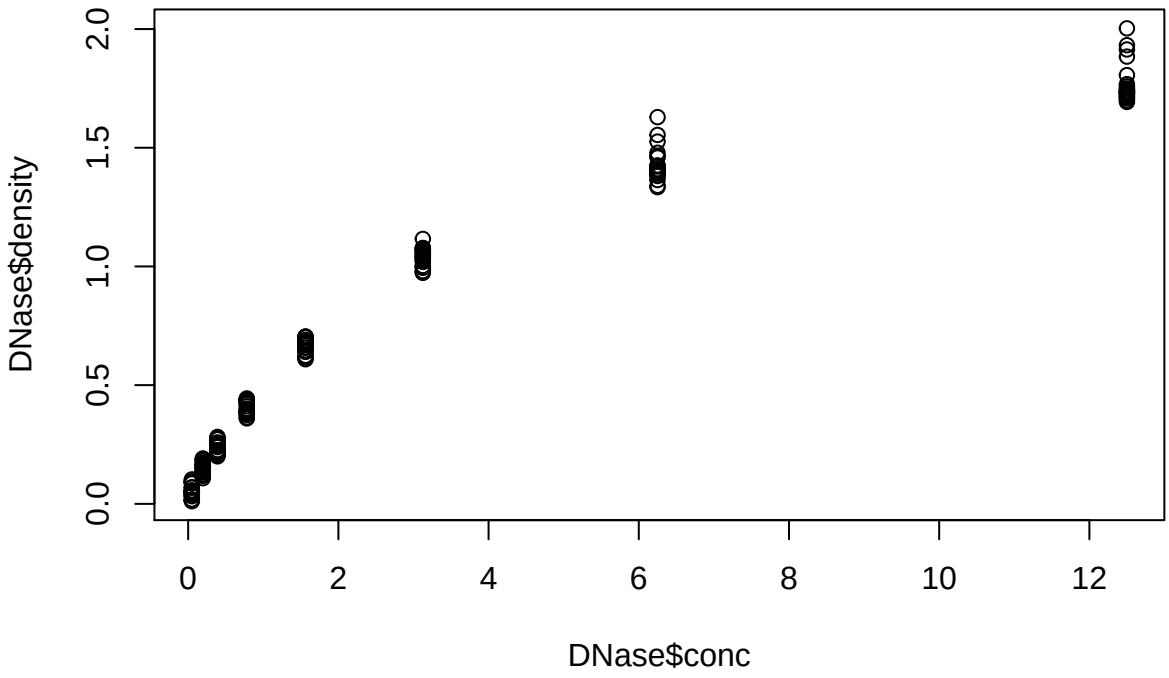
```
head(DNase)
```

```
##   Run       conc density
## 1   1 0.04882812   0.017
## 2   1 0.04882812   0.018
## 3   1 0.19531250   0.121
## 4   1 0.19531250   0.124
## 5   1 0.39062500   0.206
## 6   1 0.39062500   0.215
```

The object DNase is a dataframe whose columns are Run, the assay run; conc, the protein concentration that was used; and density, the measured optical density.

By using the `plot` function, we can easily visualize two of the variables in our data set.

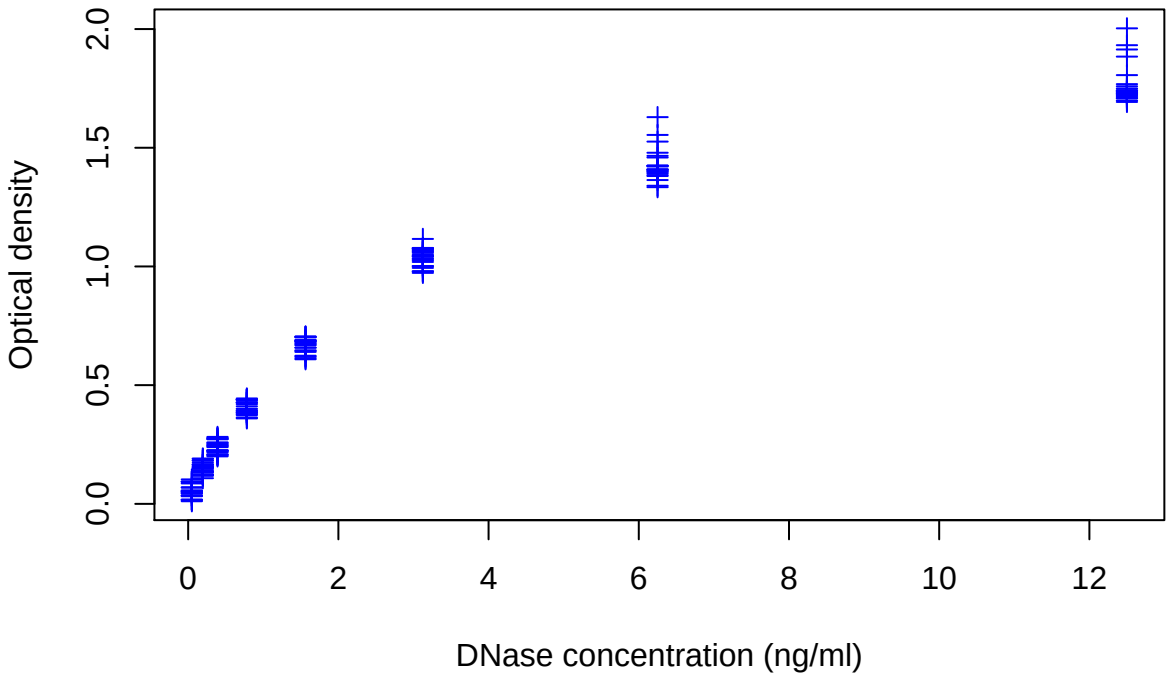
```
plot(DNase$conc, DNase$density)
```



This plot is not particularly visually appealing, but serves the purpose of quickly exploring your data.

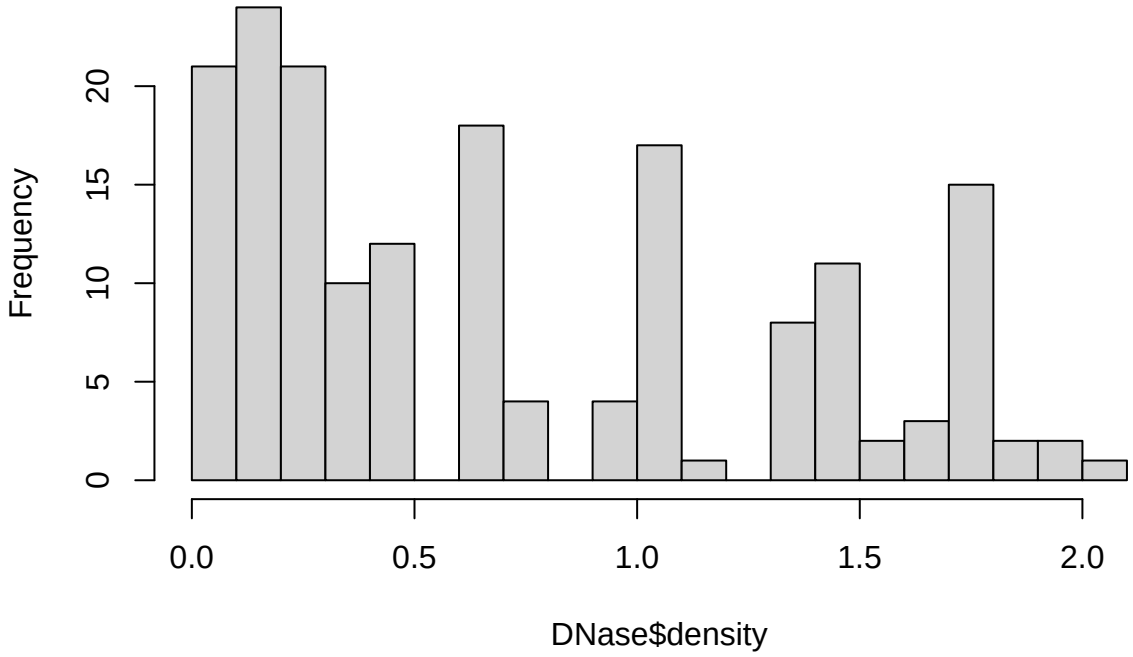
You can easily add some information to this base plot. The axis labels can be changed using the `xlab` and `ylab` arguments. `pch` deternines the type of symbol you will use for plotting (1 = circles, 2 = triangles, etc.).

```
plot(DNase$conc, DNase$density,
  ylab = attr(DNase, "labels")$y,
  xlab = paste(attr(DNase, "labels")$x, attr(DNase, "units")$x),
  pch = 3,
  col = "blue")
```



We can also use other types of plots, usually by calling them by their name.

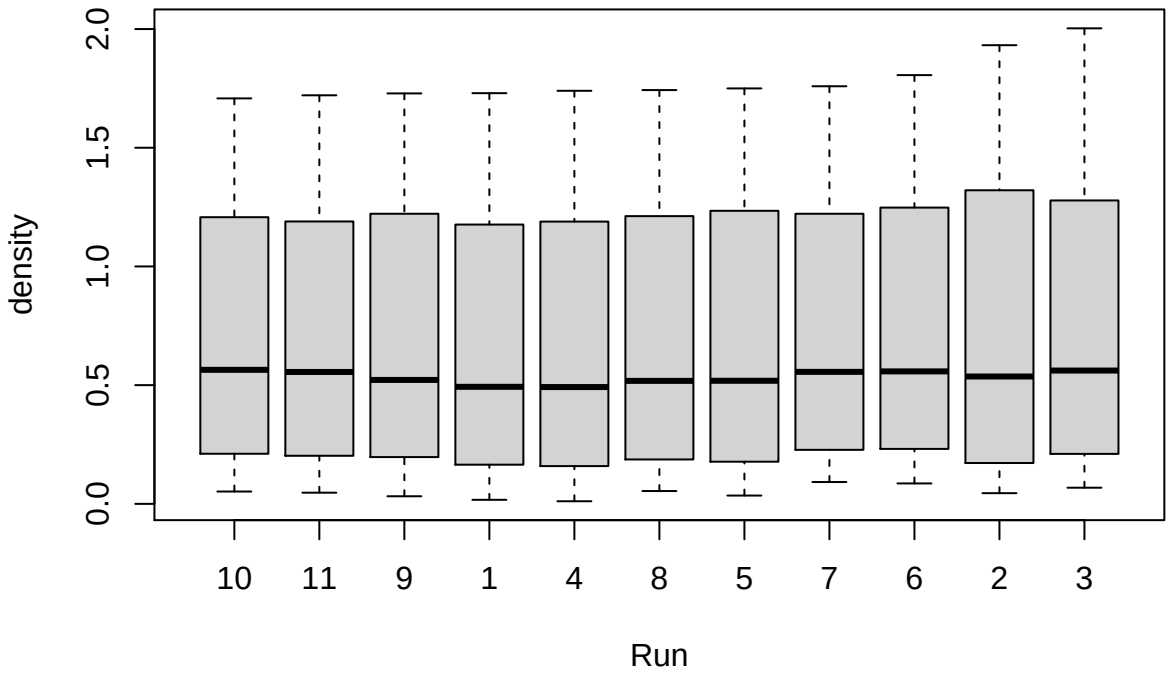
```
hist(DNase$density, breaks=25, main = "")
```



Exercise:

Create a boxplot that plots the assay runs on the x-axis and the optical density on the y-axsi!

```
boxplot(density ~ Run, data = DNase)
```



This way of displaying your data is convenient, but you will quickly reach the limits of what is possible. Especially if you want to create highly individual figures for your publications, base R plotting will often not be sufficient.

Example data set

```
library("Hiiragi2013")
data("x")
dim(Biobase::exprs(x))
```

```
## [1] 45101   101
```

ggplot2

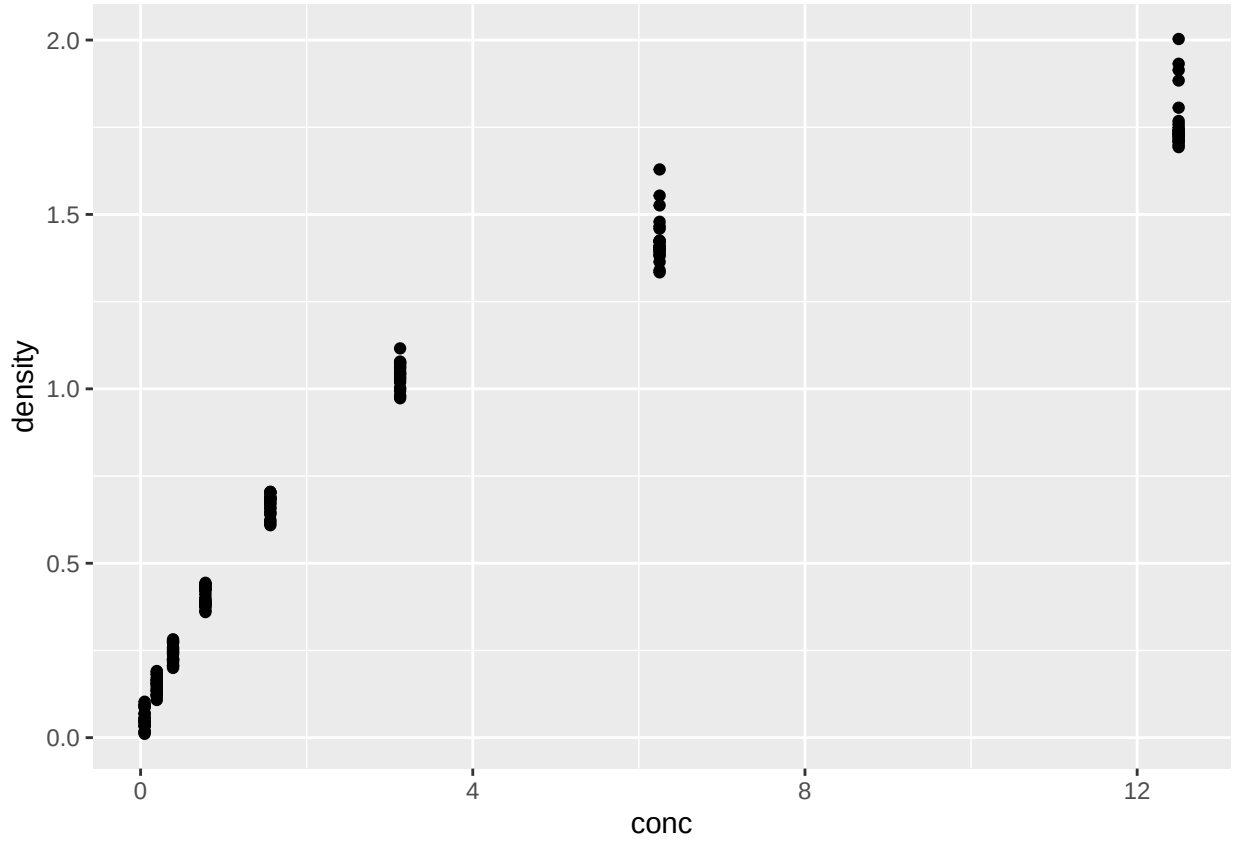
First we need to load the infamous package ggplot2.

```
library("ggplot2")
```

Next, almost as easily as in base R, we can create a plot of two variables. For this we use the ggplot2 package by Hadley Wickham which is based on the grammar of graphics. The special thing here is that we can build our plots similar to a sentence. First we specify which data we want to work with. Then we name

the aesthetics, i.e., which variables should be mapped to the x-axis or y-axis. And last but not least, we can specify the geometries, i.e., the way the data is represented in the coordinate system.

```
ggplot(DNase, aes(x = conc, y = density)) + geom_point()
```



Again, we can take a look at the first six rows of the example data set using the `head` function. Inside the `head` function, we apply the `pData` function which retrieves meta-data from an expression data set. In this case, we retrieve phenotypic data.

```
head(pData(x), n = 2)
```

```
##         File.name Embryonic.day Total.number.of.cells lineage genotype
## 1 E3.25  1_C32_IN         E3.25                    32               WT
## 2 E3.25  2_C32_IN         E3.25                    32               WT
##           ScanDate sampleGroup sampleColour
## 1 E3.25 2011-03-16       E3.25      #CAB2D6
## 2 E3.25 2011-03-16       E3.25      #CAB2D6
```

One of the most important aspects in data visualization is the appropriate preparation of the data set in R. A package that can be particularly helpful is dplyr.

```
library("dplyr")
```

Using the `group_by` function from the dyplr package, you can group your data frame by one or more variables. Next, we pipe the output of the `group_by` function into the `summarise` function which will summarise the grouped data frame. More specifically, we use the `n` function which gives the size of each group.

```
groups = group_by(pData(x), sampleGroup) %>%
  summarise(n = n(), color = unique(sampleColour))
```

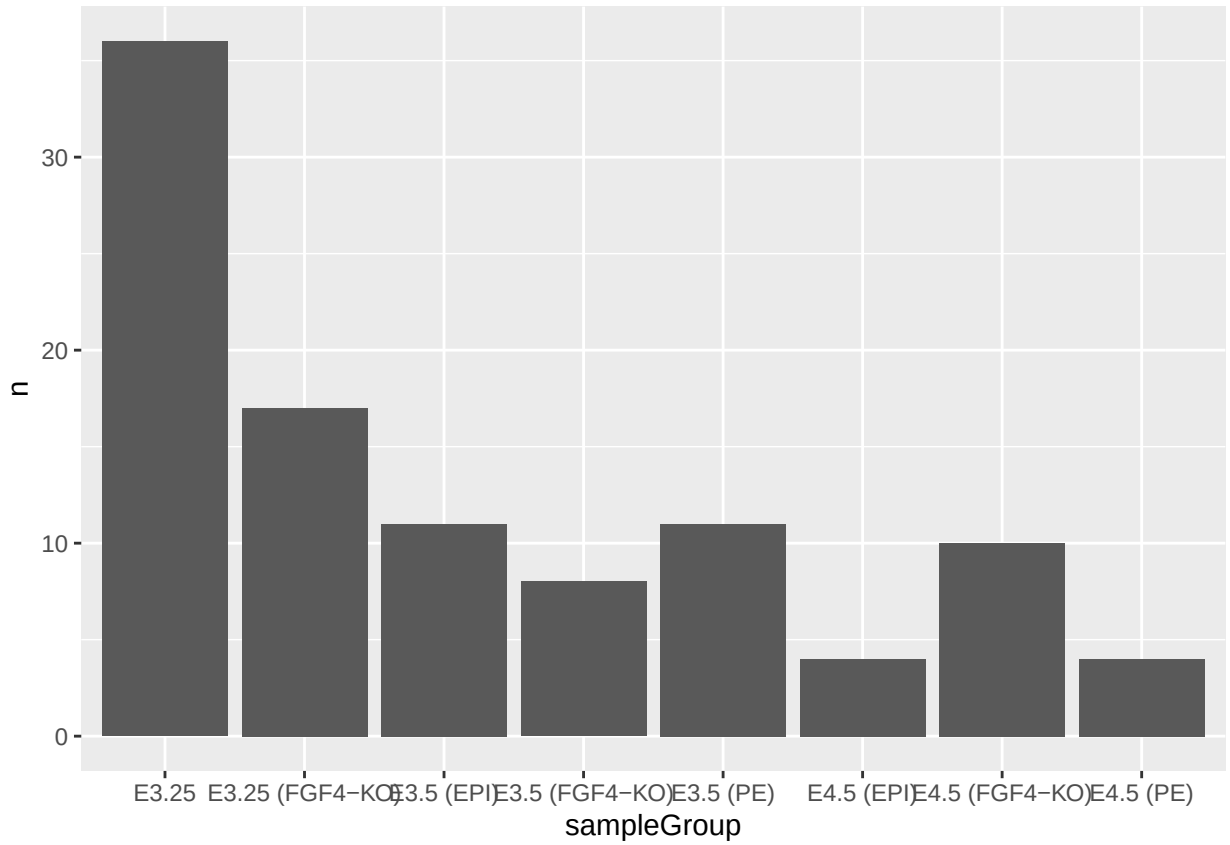
```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
groups
```

```
## # A tibble: 8 x 3
##   sampleGroup          n color
##   <chr>            <int> <chr>
## 1 E3.25               36 #CAB2D6
## 2 E3.25 (FGF4-KO)     17 #FDBF6F
## 3 E3.5 (EPI)          11 #A6CEE3
## 4 E3.5 (FGF4-KO)       8 #FF7F00
## 5 E3.5 (PE)           11 #B2DF8A
## 6 E4.5 (EPI)           4 #1F78B4
## 7 E4.5 (FGF4-KO)      10 #E31A1C
## 8 E4.5 (PE)            4 #33A02C
```

We can now plot the newly created grouped variables in a bar plot by specifying the corresponding variables and the `geom_bar` geometry.

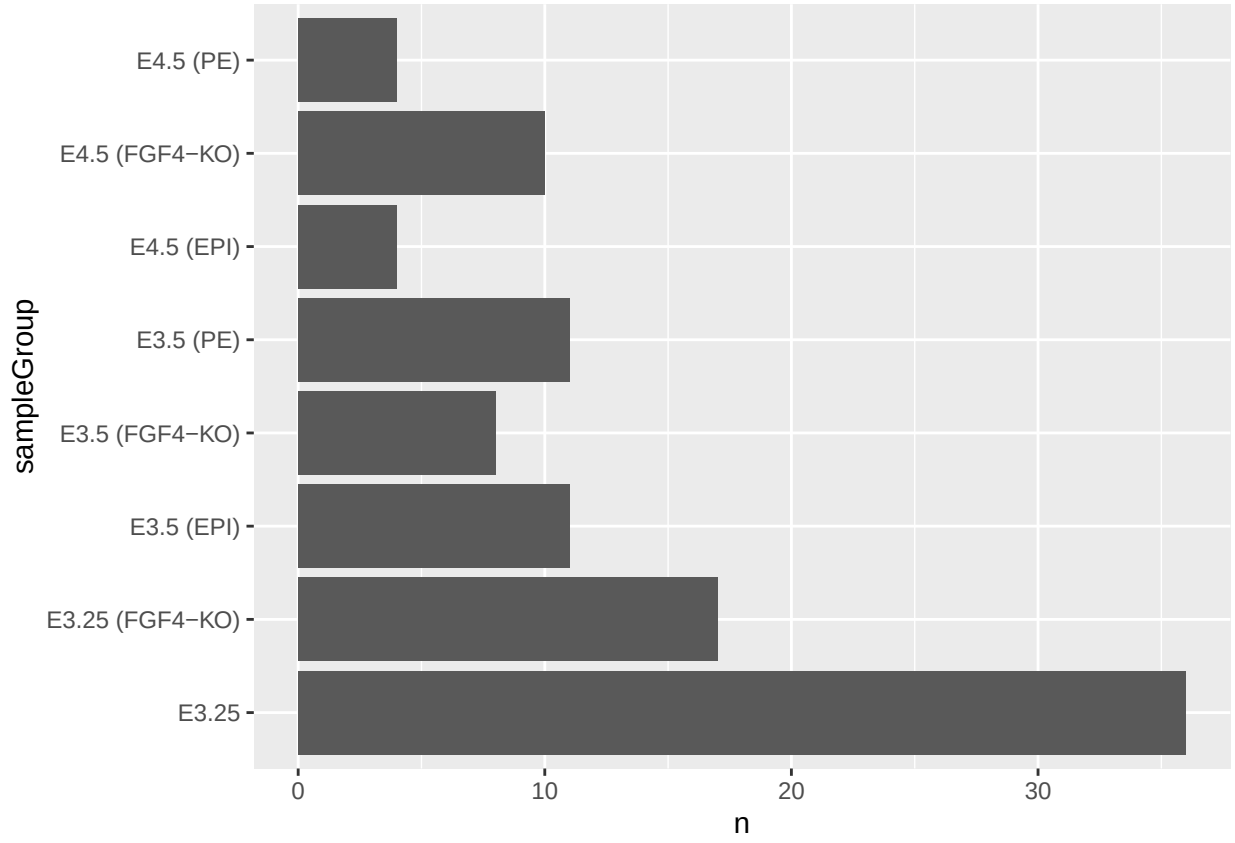
```
ggplot(groups, aes(x = sampleGroup, y = n)) +
  geom_bar(stat = "identity")
```



Exercise:

Try to produce a horizontal barplot!

```
ggplot(groups, aes(x = n, y = sampleGroup)) +
  geom_bar(stat = "identity")
```



In your publications, it makes a good impression if you use certain color schemes in a consistent manner throughout, with different groups always having the same color. To achieve this, we use the vector with colors already automatically generated by the `group_by` function.

```
groupColor = setNames(groups$color, groups$sampleGroup)
```

If you want to make your plots colorblind friendly, I can recommend the following series of blog posts.

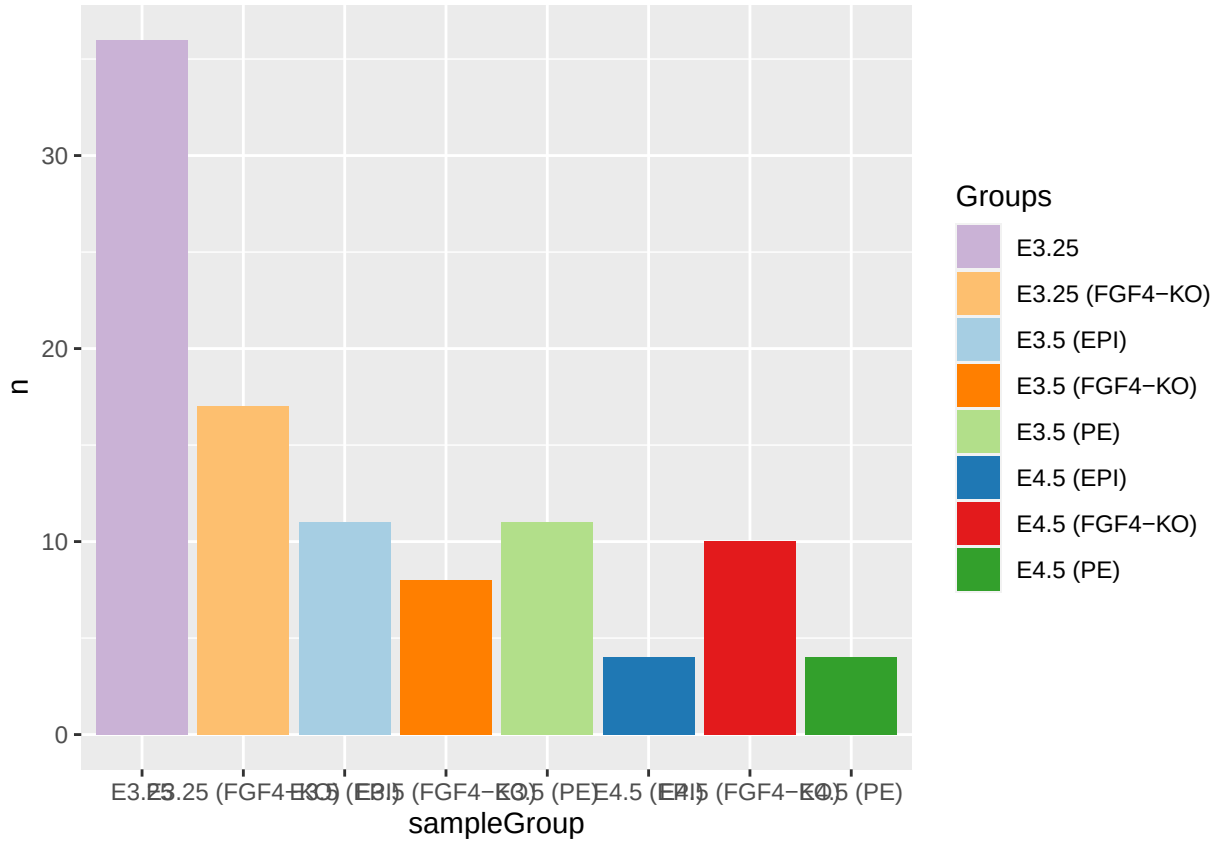
https://blog.datawrapper.de/colorblindness-part1/

Also, you can have a look at the `colorblindr` package

remotes::install_github("wilkelab/cowplot") install.packages("colorspace", repos = "http://R-Forge.R-project.org")

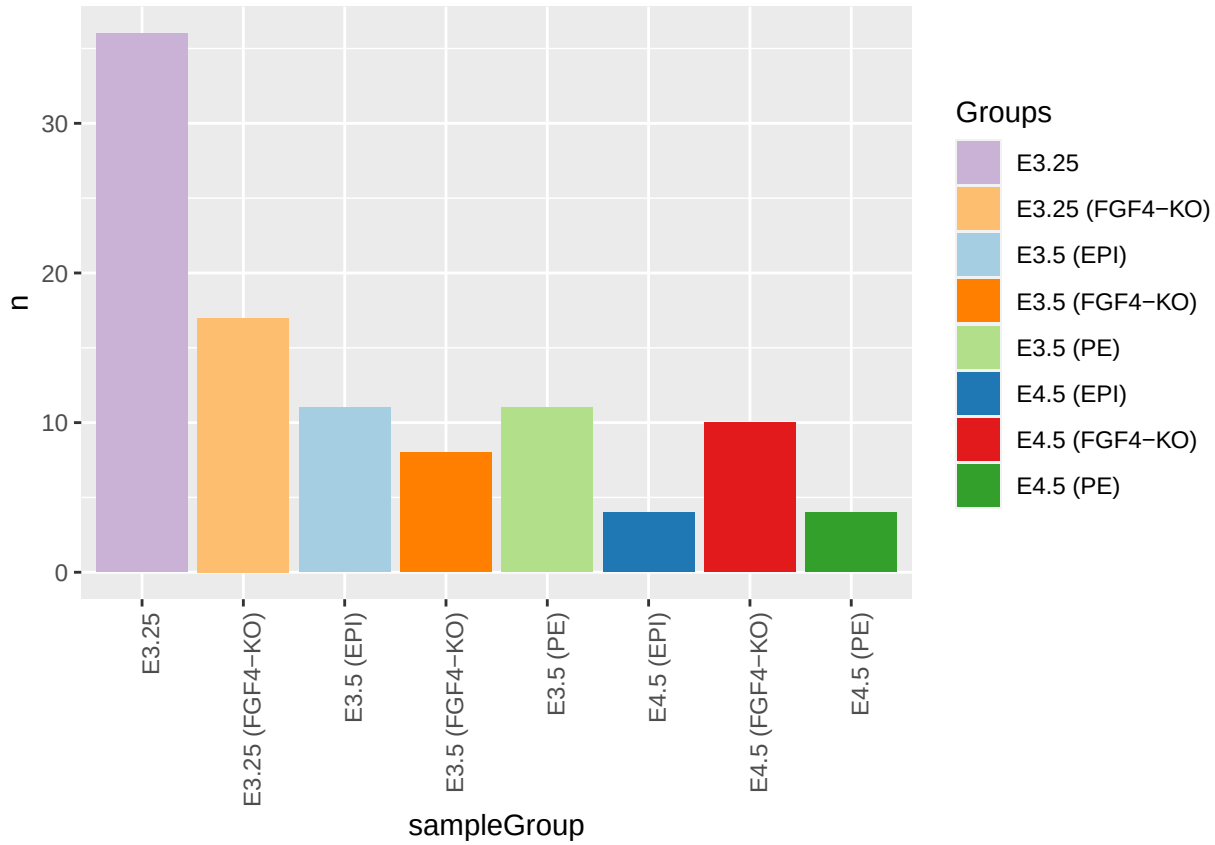
We can add the colors by first adding the `fill` argument to the aesthetics function which will fill the bars with color. To manually set the colors we can add another layer to our plot using the `+` symbol and the `scale_fill_manual` function.

```
p <- ggplot(groups, aes(x = sampleGroup, y = n, fill = sampleGroup)) +
        geom_bar(stat = "identity") +
        scale_fill_manual(values = groupColor, name = "Groups")
p
```



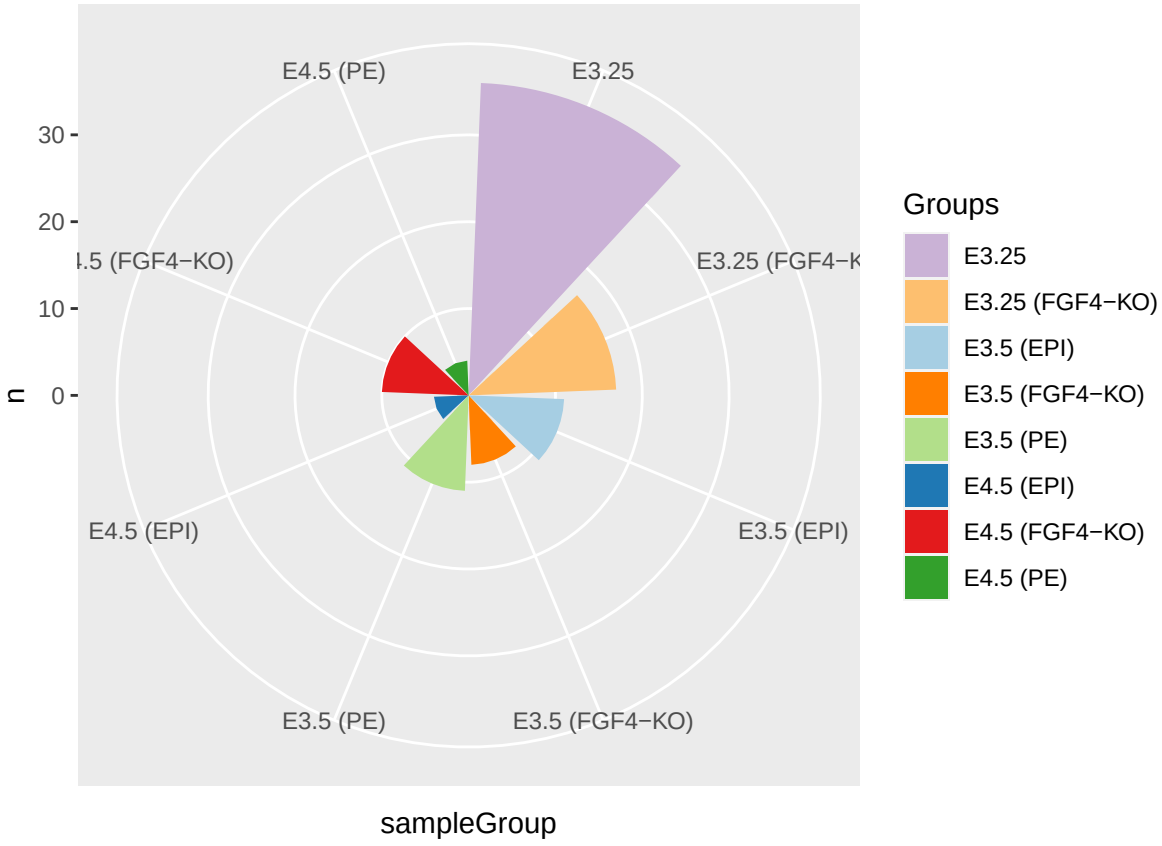
Now, we still have the problem of overlapping bar labels. By adding another layer, the `theme` function, we can costumize non-data components of our plot and again easily add them using `+`. Here, we change the angle of the axis text on the x-axis.

```
p + theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

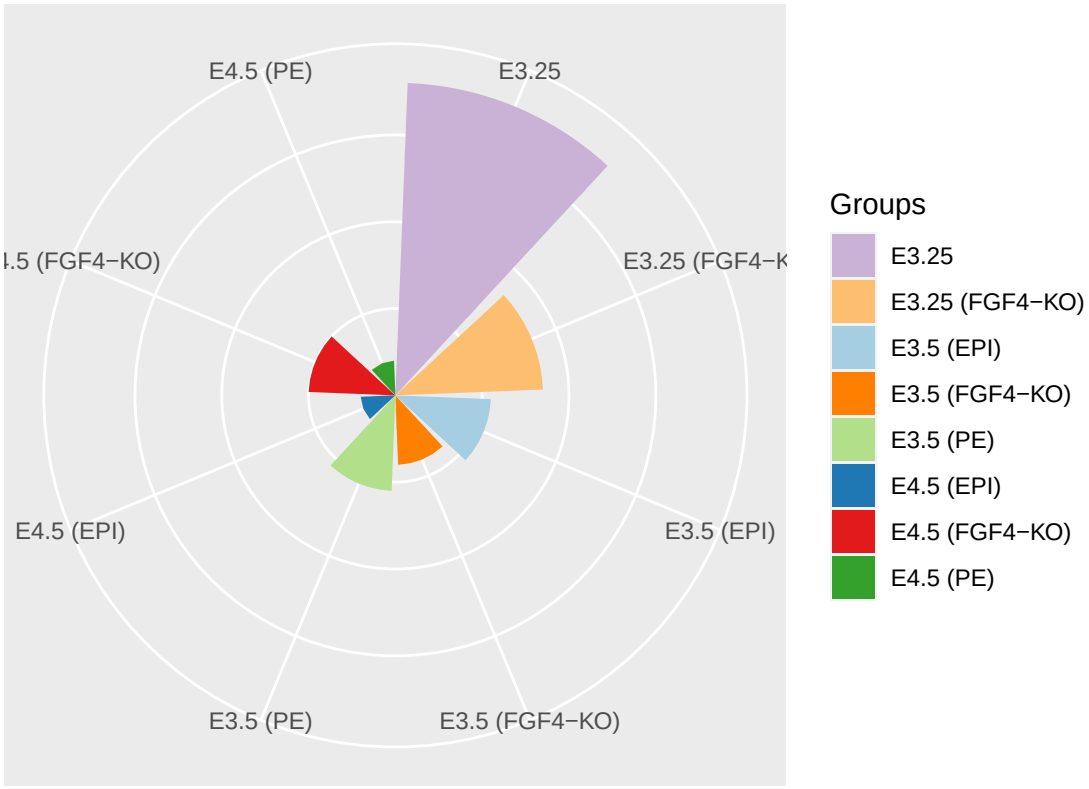


This modularized way of designing figures makes it easy to test different visualization ideas without having to start from scratch with the creation of the figure each time.

```
p + coord_polar()
```




```
p_polar = p + coord_polar() +
  theme(axis.text.x = element_text(angle = 0, hjust = 1),
        axis.text.y = element_blank(),
        axis.ticks = element_blank()) +
  xlab("") + ylab("")
p_polar
```



We can save our figures using the `ggsave` function.

Grammar of graphics

Components of ggplot´s grammar of graphics:

1. one or more data sets
2. one or more geometric objects that serve as the visual representations of the data, – for instance, points, lines, rectangles, contours,
3. descriptions of how the variables in the data are mapped to visual properties (aesthetics) of the geometric objects, and an associated scale (e. g., linear, logarithmic, rank),

Optional:

4. one or more coordinate systems,
5. statistical summarization rules,
6. a facet specification, i.e. the use of multiple similar subplots to look at subsets of the same data,
7. optional parameters that affect the layout and rendering, such text size, font and alignment, legend positions.

(see Holmes and Huber 2018, ch. 3.5)

- These concepts –data, geometrical objects, statistics– are some of the ingredients of the grammar of graphics, just as nouns, verbs and adverbs are ingredients of an English sentence.

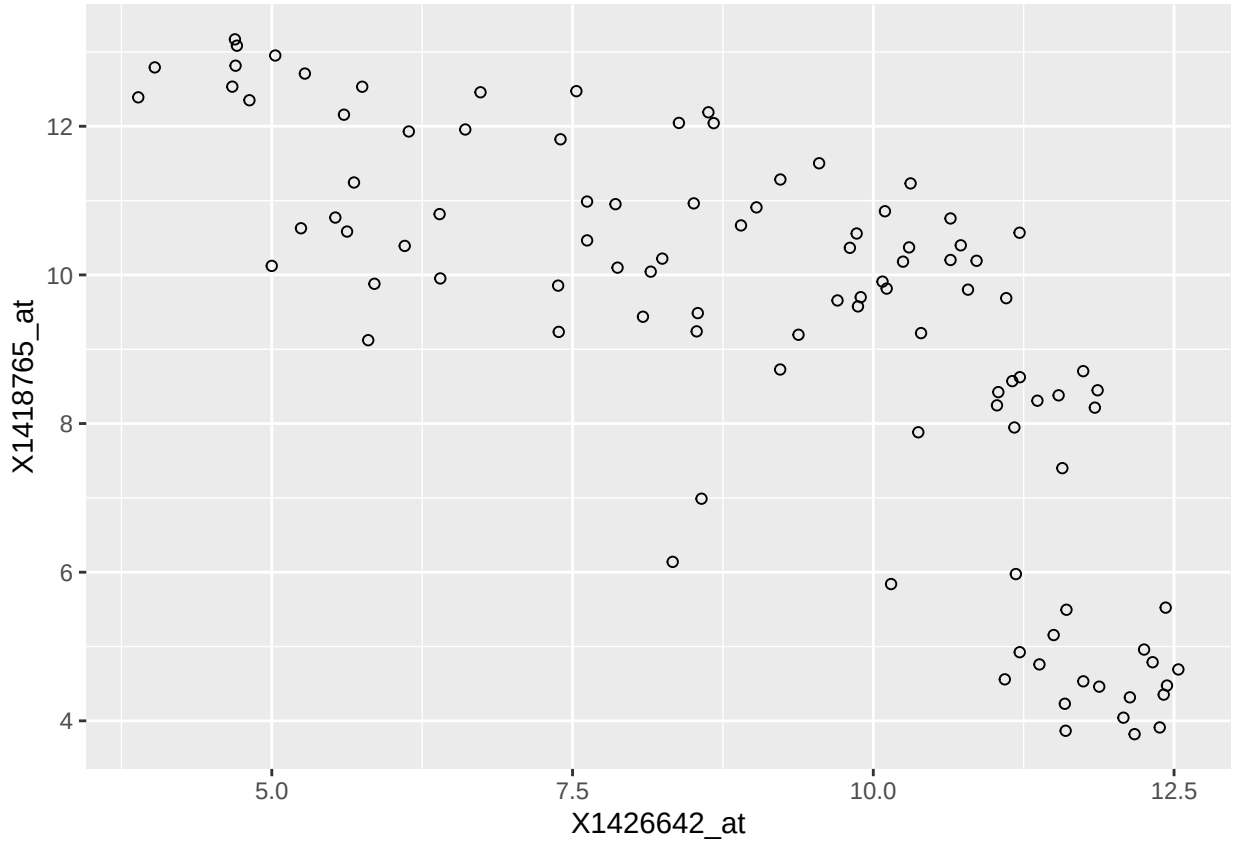
```
dftx = data.frame(t(Biobase::exprs(x)), pData(x))
```

To illustrate the structure of ggplot2, we will build a plot layer by layer. First we start with a blank canvas.

```
ggplot()
```


As specified by the components of the grammar of graphics, we need at least the first three components to create a meaningful figure.

```
ggplot( dftx, aes( x = X1426642_at, y = X1418765_at)) +
  geom_point( shape = 1 )
```

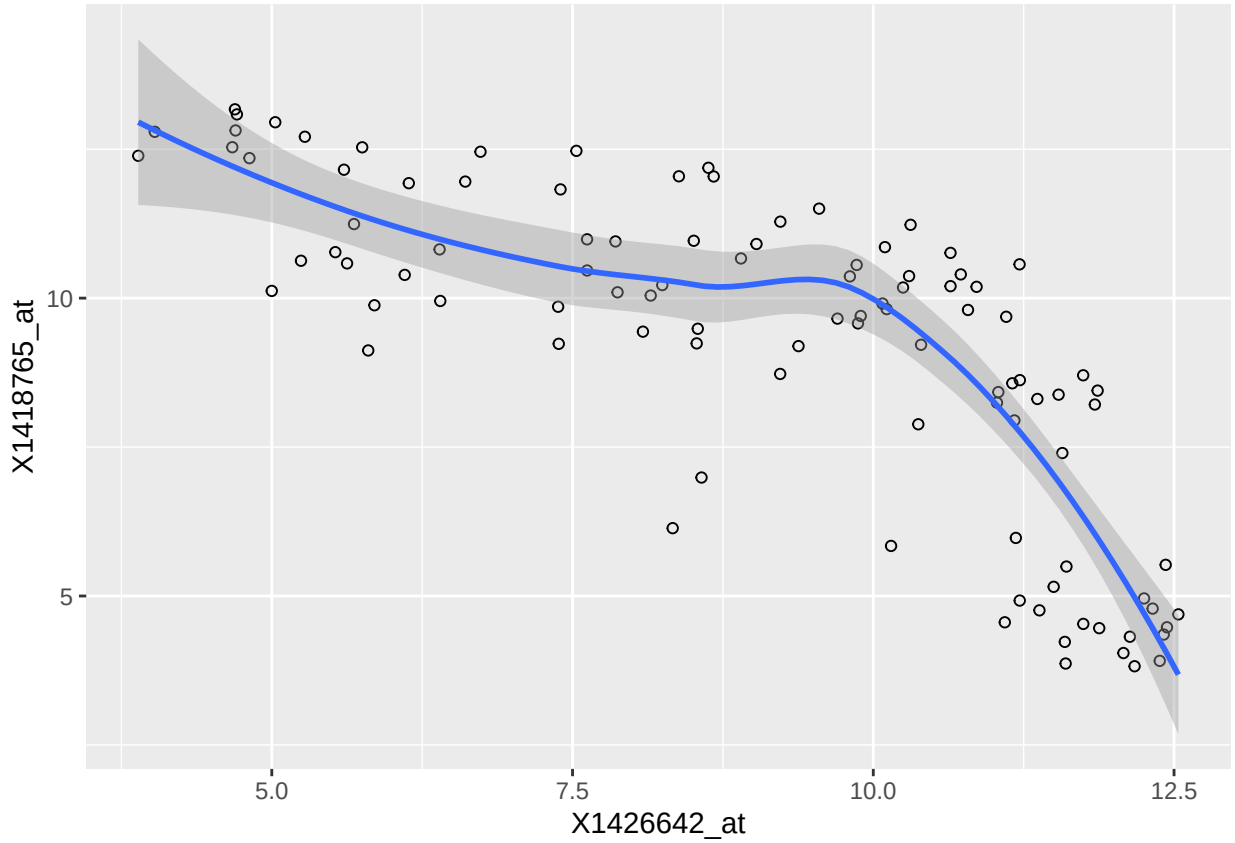


Now that we have determined a data set, a description of which variables should be mapped and how, and our geometric object, we get at least an acceptable figure.

To describe the trend of the observations one can add another layer with a local regression line using the `geom_smooth` function.

```
ggplot( dftx, aes( x = X1426642_at, y = X1418765_at)) +
  geom_point( shape = 1 ) +
  geom_smooth( method = "loess" )
```

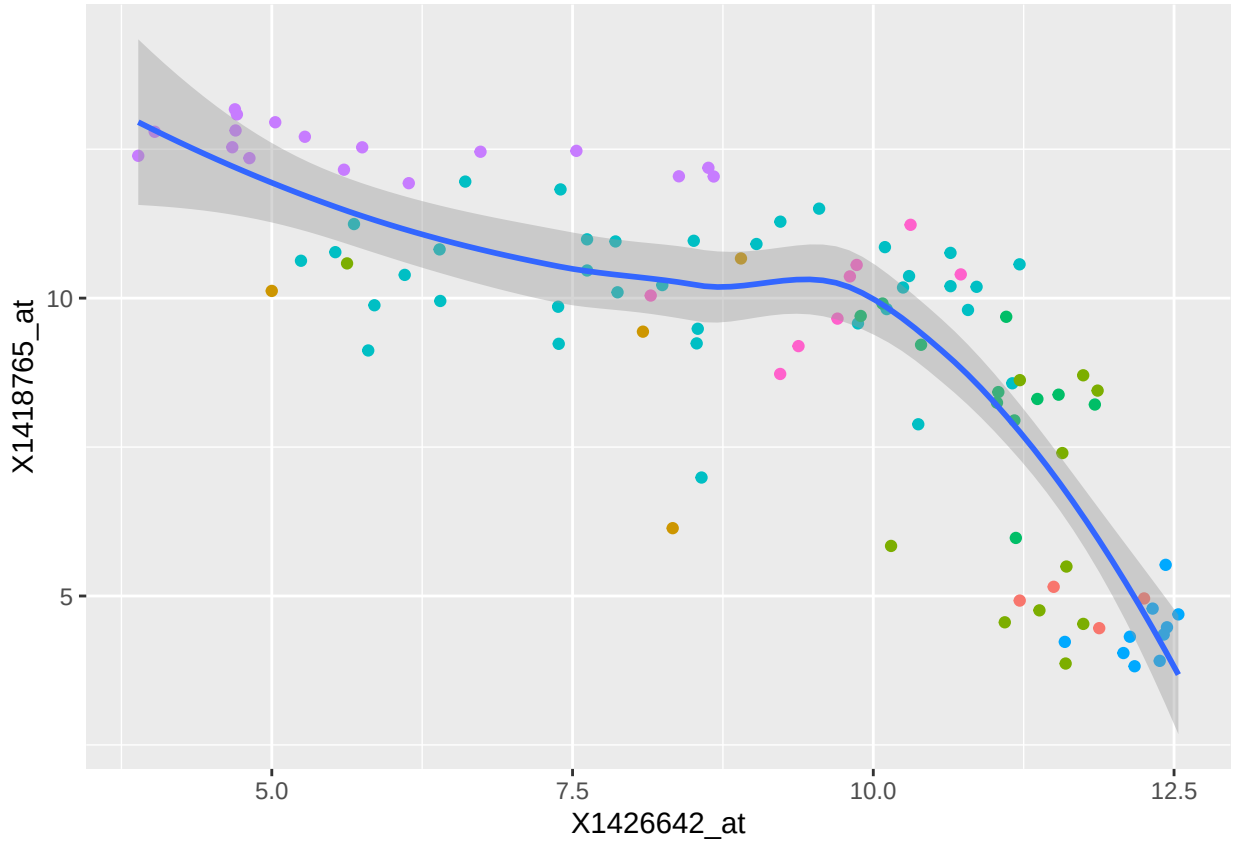
```
## `geom_smooth()` using formula 'y ~ x'
```



As you can see, you can create increasingly complex figures by adding information layer by layer.

```
ggplot( dftx, aes( x = X1426642_at, y = X1418765_at ))  +
  geom_point( aes( color = sampleColour), shape = 19 ) +
  geom_smooth( method = "loess" ) +
  scale_color_discrete( guide = FALSE )
```

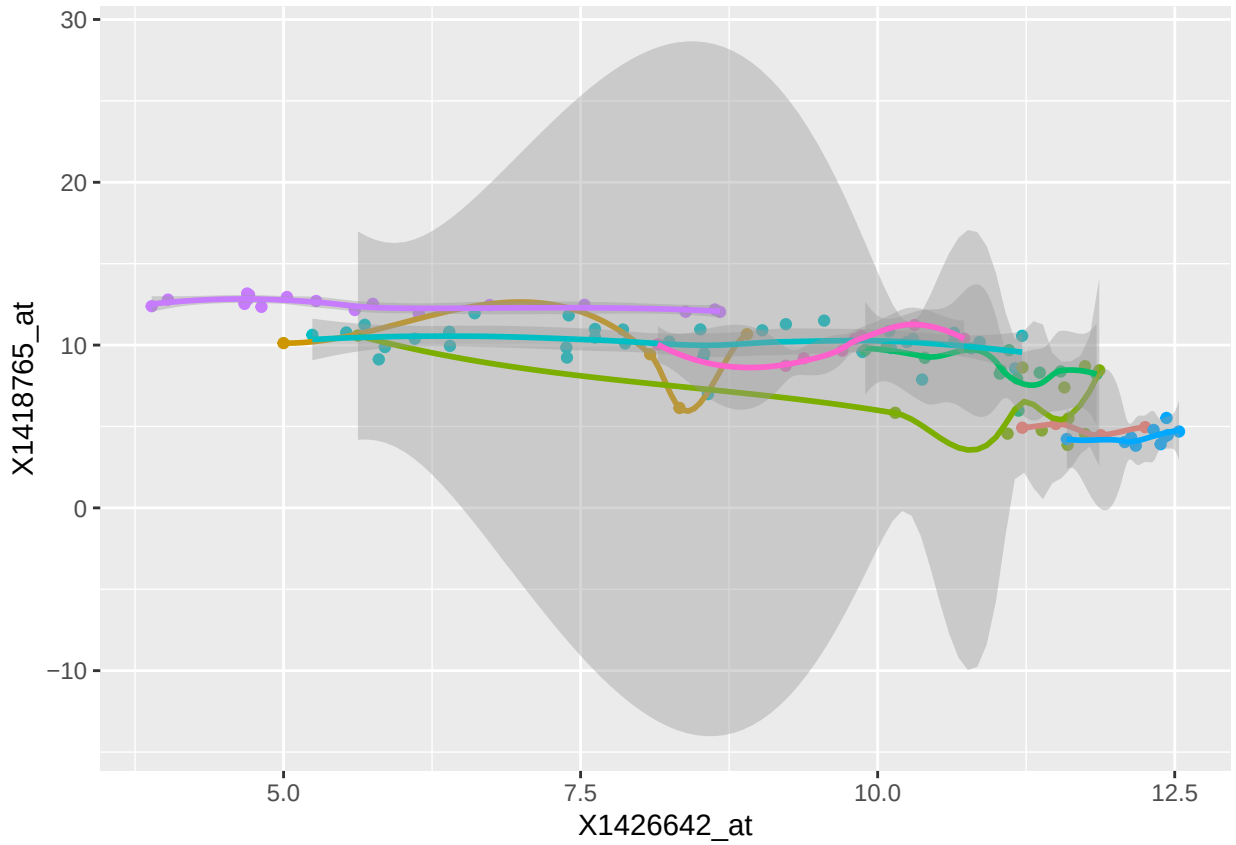
```
## `geom_smooth()` using formula 'y ~ x'
```



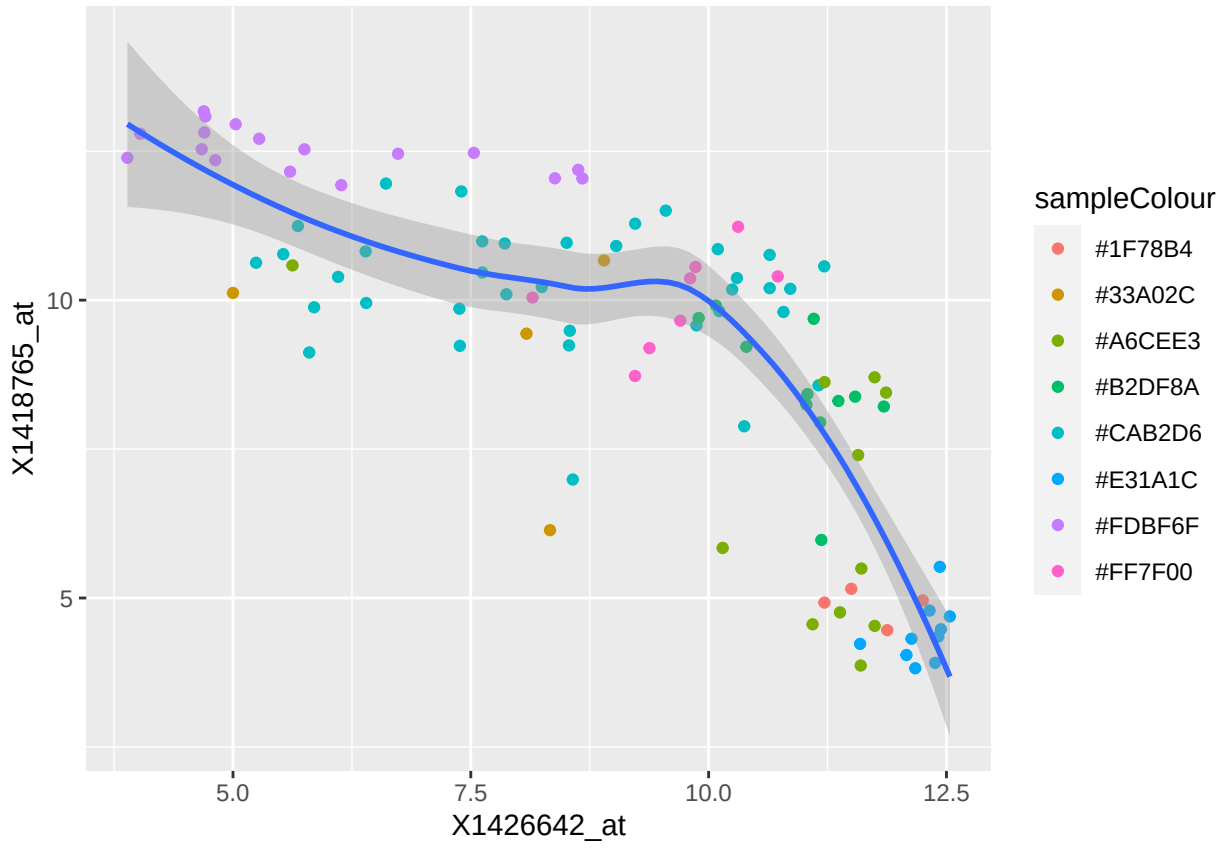
Exercise:

In the code above we defined the color aesthetics (aes) only for the geom_point layer, while we defined the x and y aesthetics for all layers. What happens if we set the color aesthetics for all layers, i.e., move it into the argument list of ggplot? What happens if we omit the call to scale_color_discrete?

```
ggplot( dftx, aes( x = X1426642_at, y = X1418765_at, color = sampleColour ))  +
  geom_point( shape = 19 ) +
  geom_smooth( method = "loess" ) +
  scale_color_discrete( guide = FALSE )
```




```
ggplot( dftx, aes( x = X1426642_at, y = X1418765_at ))  +
  geom_point( aes( color = sampleColour), shape = 19 ) +
  geom_smooth( method = "loess" )
```




```
library("mouse4302.db")
```

```
AnnotationDbi::select(mouse4302.db,
   keys = c("1426642_at", "1418765_at"), keytype = "PROBEID",
   columns = c("SYMBOL", "GENENAME"))
```

```
## 'select()' returned 1:1 mapping between keys and columns
```

```
##      PROBEID SYMBOL                                              GENENAME
## 1 1426642_at    Fn1                                          fibronectin 1
## 2 1418765_at  Timd2 T cell immunoglobulin and mucin domain containing 2
```

```
#ggsave("DNAse-histogram-demo.pdf", plot = p)
```

Directory of Visualizations

If you plan to visualize something, it makes sense to consider beforehand whether you want to display your data one-dimensionally or two-dimensionally.

For me, the book "Fundamentals of Data Visualization" by Claus O. Wilke has proven to be the best choice when considering which type of visualization to use.

You can find the complete Version of this book here. If you want to adapt or reproduce one of the figures from the book in R, you can have a look at the Github repository here.

First, I usually open Chapter 5, Directory of Visualizations, in Fundamentals of Data Visualization and choose between the following visualizations for different data types:

- Amounts
- Distributions
- Proportions
- x–y relationships
- Geospatial Data
- Uncertainty

After I have decided on a visualization form, I go to the corresponding chapter in the book and take a closer look at what needs to be considered for this data type.

In the following we will present some examples of visualizations.

Wilke (2019)

Visualizing data in 1D

```
selectedProbes = c( Fgf4 = "1420085_at", Gata4 = "1418863_at",
                    Gata6 = "1425463_at",  Sox2 = "1416967_at")
```

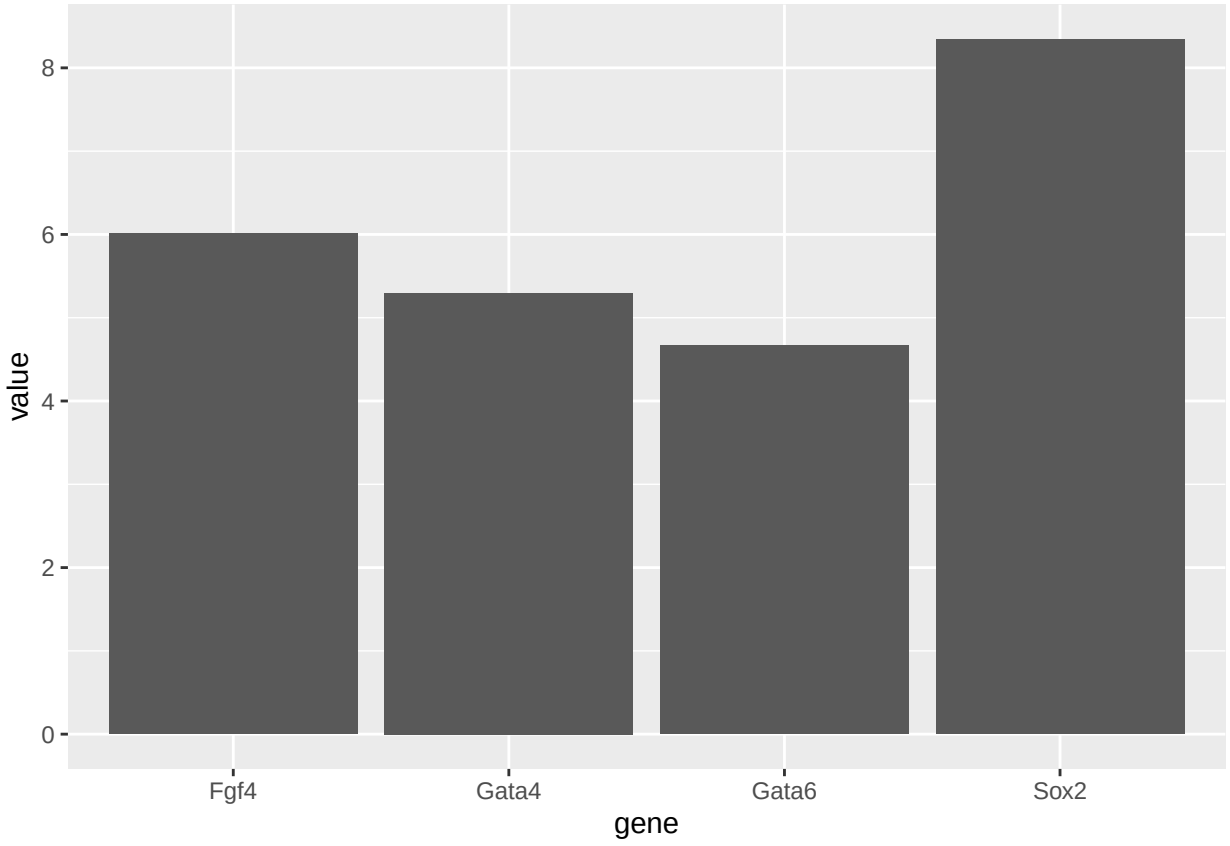
```
library("reshape2")
genes = melt(Biobase::exprs(x)[selectedProbes, ],
             varnames = c("probe", "sample"))
head(genes)
```

```
##        probe  sample    value
## 1 1420085_at 1 E3.25 3.027715
## 2 1418863_at 1 E3.25 4.843137
## 3 1425463_at 1 E3.25 5.500618
## 4 1416967_at 1 E3.25 1.731217
## 5 1420085_at 2 E3.25 9.293016
## 6 1418863_at 2 E3.25 5.530016
```

```
genes$gene = names(selectedProbes)[match(genes$probe, selectedProbes)]
```

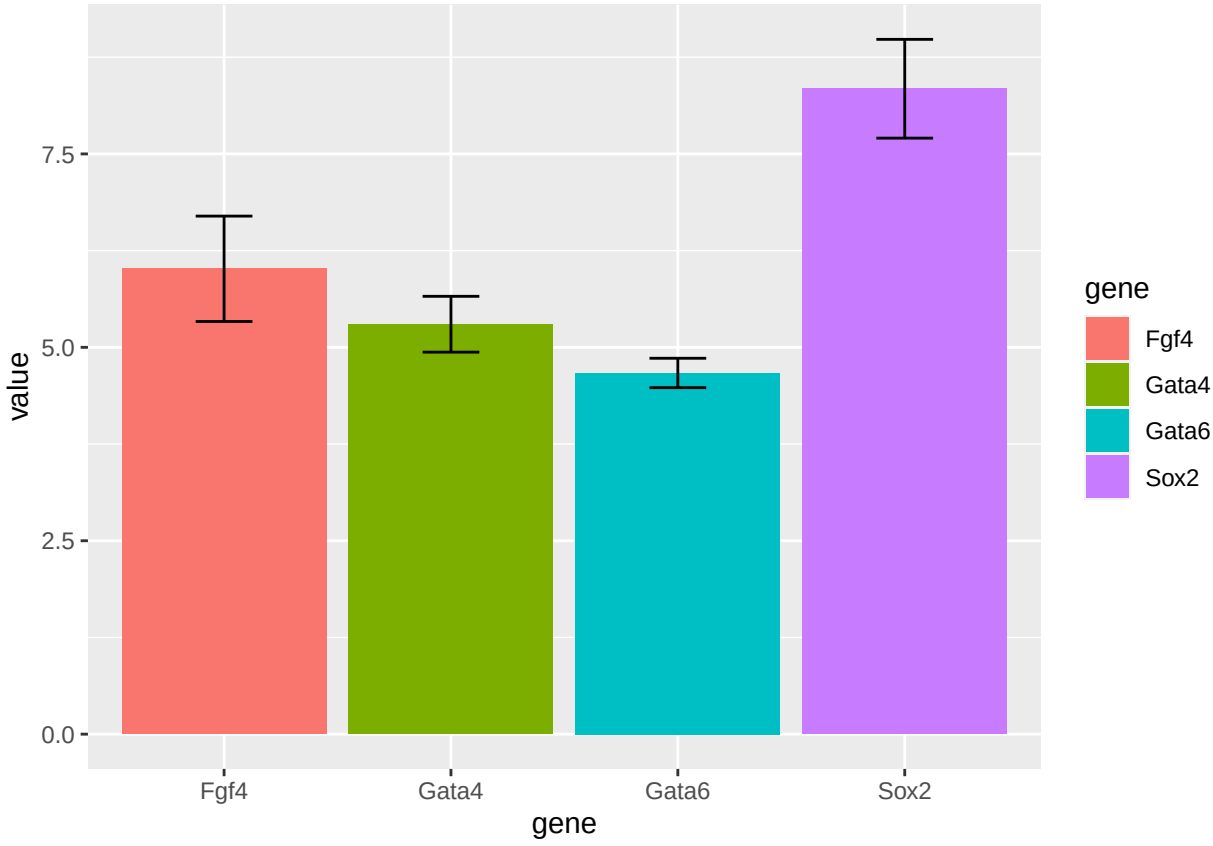
Barplots

```
ggplot(genes, aes( x = gene, y = value)) +
  stat_summary(fun.y = mean, geom = "bar")
```




```
library("Hmisc")
```

```
ggplot(genes, aes( x = gene, y = value, fill = gene)) +
  stat_summary(fun = mean, geom = "bar") +
  stat_summary(fun.data = mean_cl_normal, geom = "errorbar",
               width = 0.25)
```

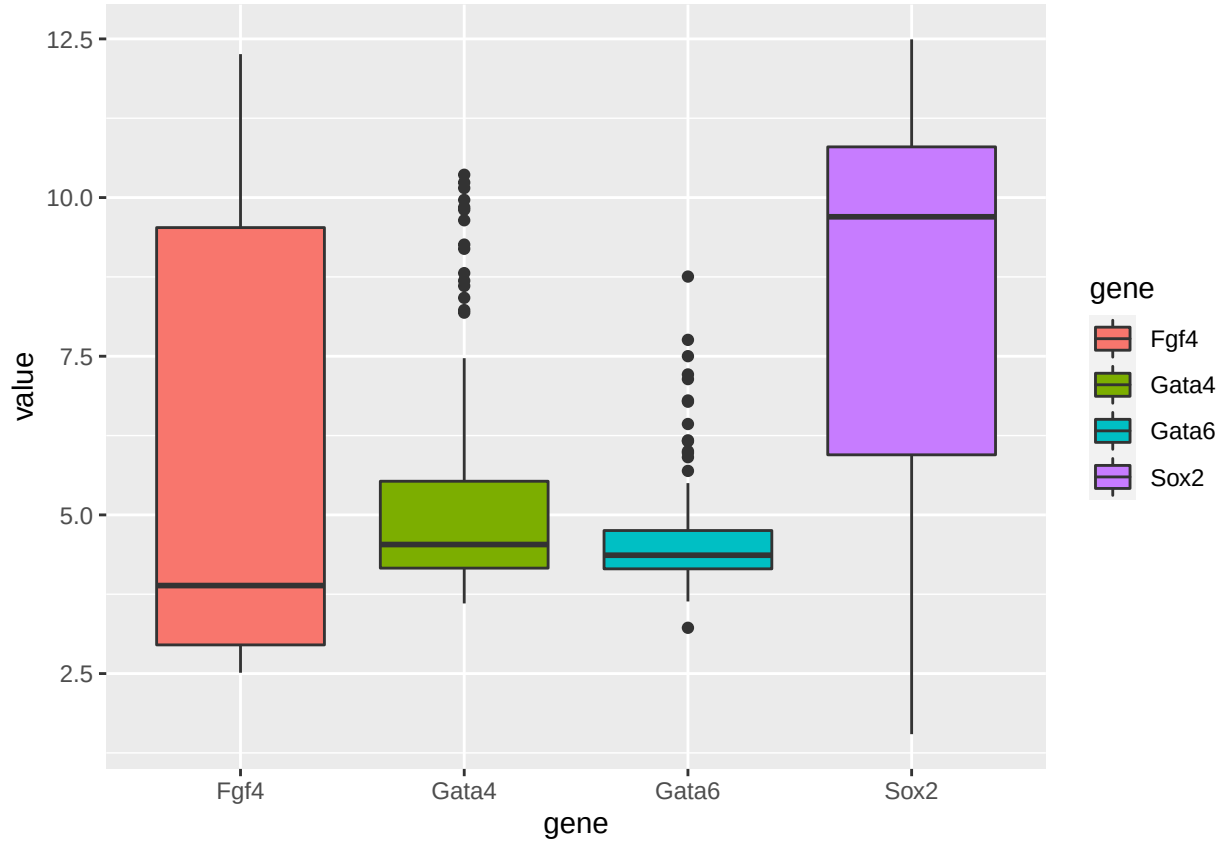


Exercise:

Create boxplot and violin plots of the same data as we used for the barplots!

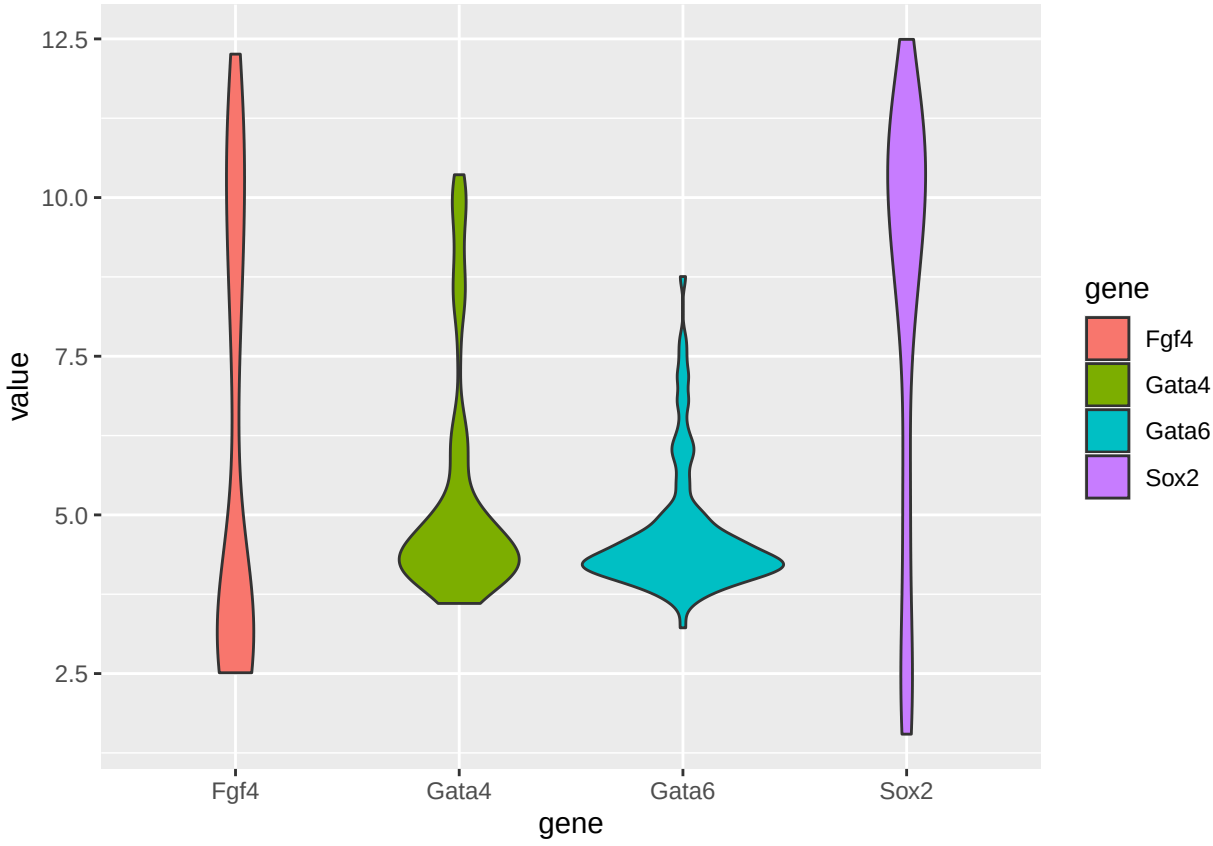
Boxplots

```
p = ggplot(genes, aes( x = gene, y = value, fill = gene))
p + geom_boxplot()
```



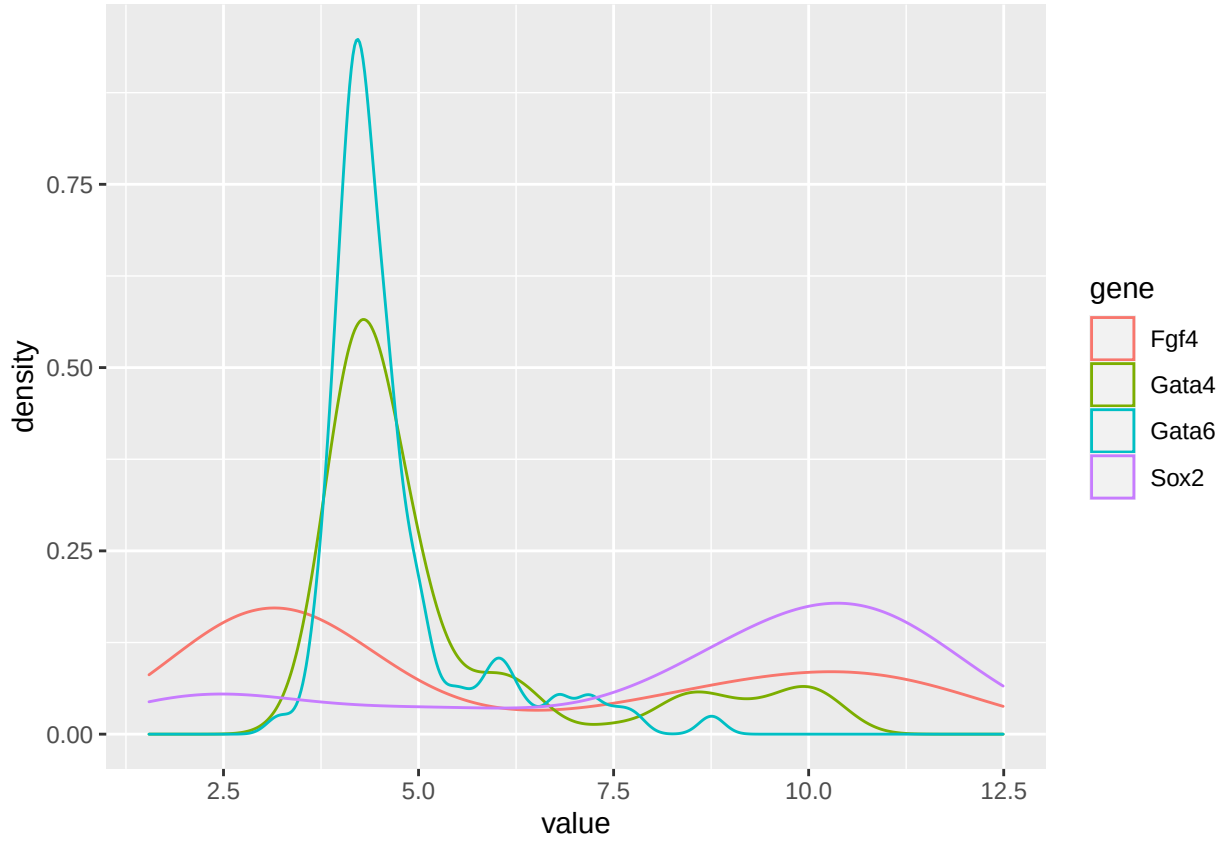
Violin plots

```
p + geom_violin()
```



Density plots

```
ggplot(genes, aes( x = value, color = gene)) + geom_density()
```



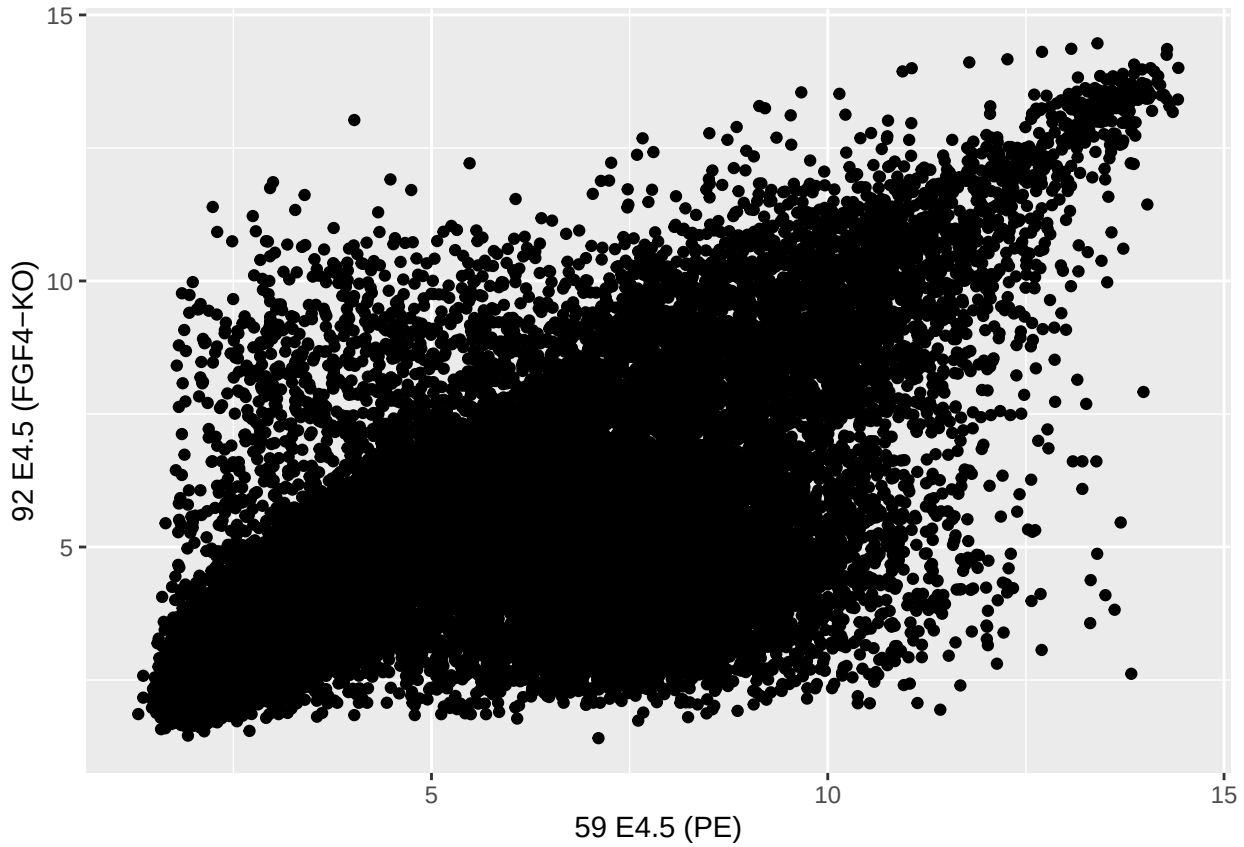
Visualizing data in 2D: scatterplots

If you want to look at the relationship between two variables, or see how a treatment and a response are related, you can use two-dimensional plots.

As an example we can look at adifferential expression between a wildtype and an FGF4-KO sample (Holmes and Huber (2018)).

```
dfx = as.data.frame(Biobase::exprs(x))
```

```
scp = ggplot(dfx, aes(x = `59 E4.5 (PE)` ,
                      y = `92 E4.5 (FGF4-KO)`))
scp + geom_point()
```



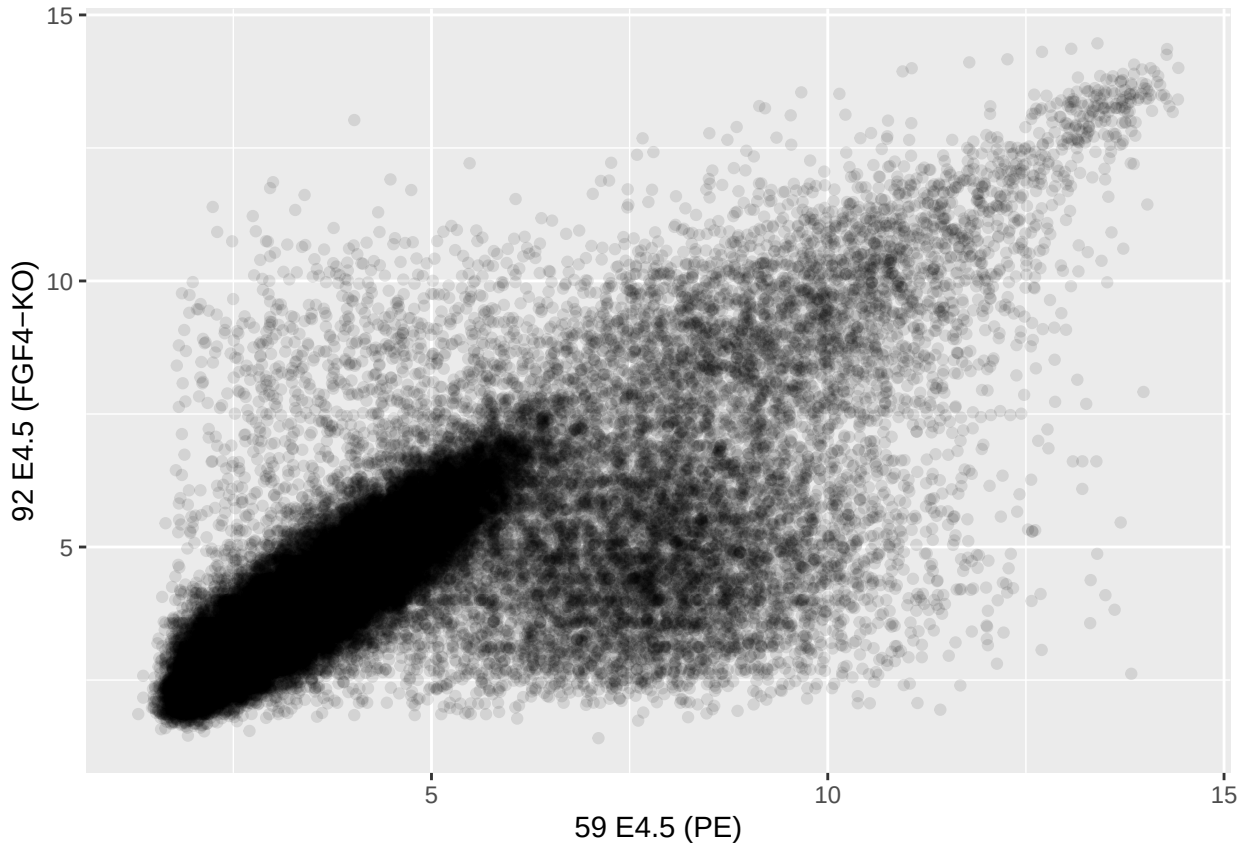
Since this figure shows too many points on top of each other, we have to think of a way to better show the distribution of the data.

Exercise:

Change the transparency of the dots and see whether it improves the visualization.

One way to do this is to adjust the transparency of the dots by changing the `alpha` value.

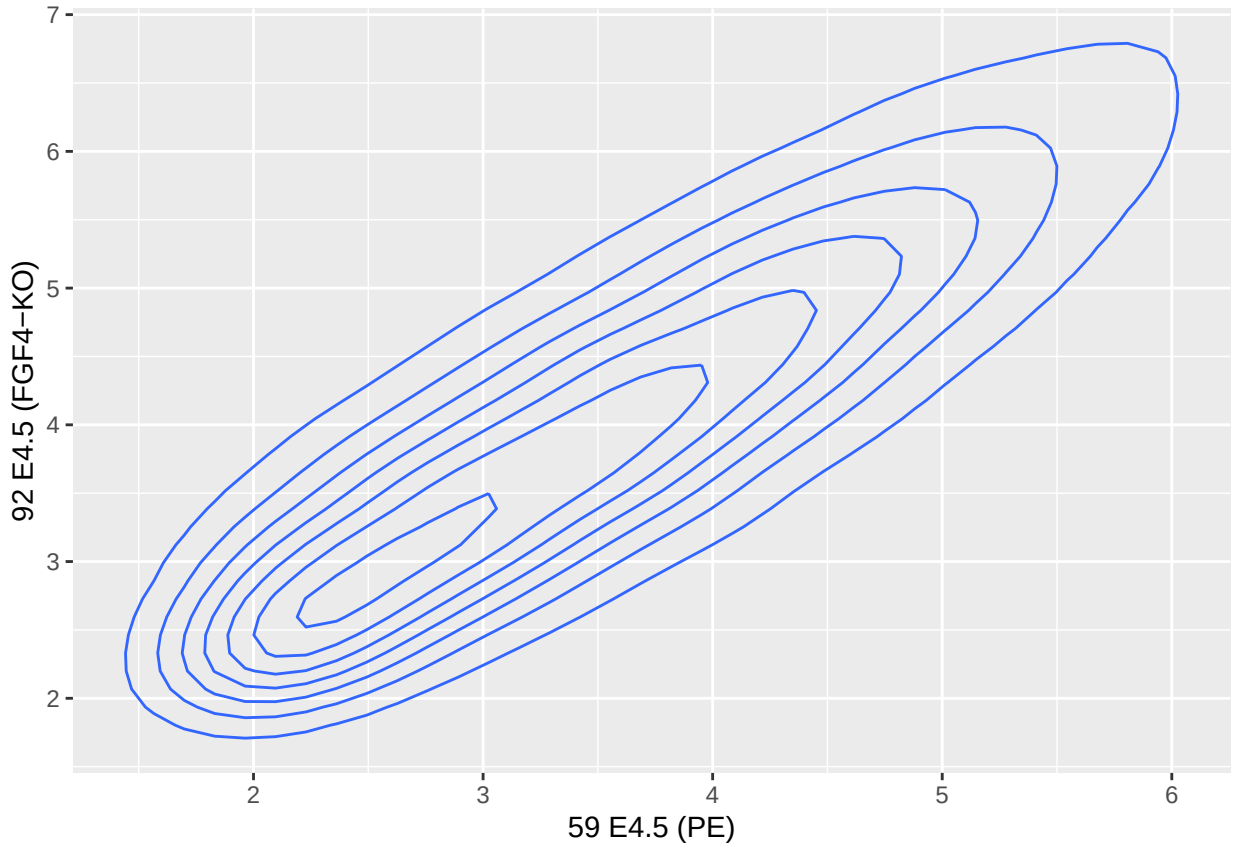
```
scp  + geom_point(alpha = 0.1)
```



Since we continue to see too much overlap despite the transparency of the points, we need to consider another option.

We can use a two-dimensional density plot instead. However, this may discard too much information and therefore does not provide an optimal solution in its default setting.

```
scp + geom_density2d()
```



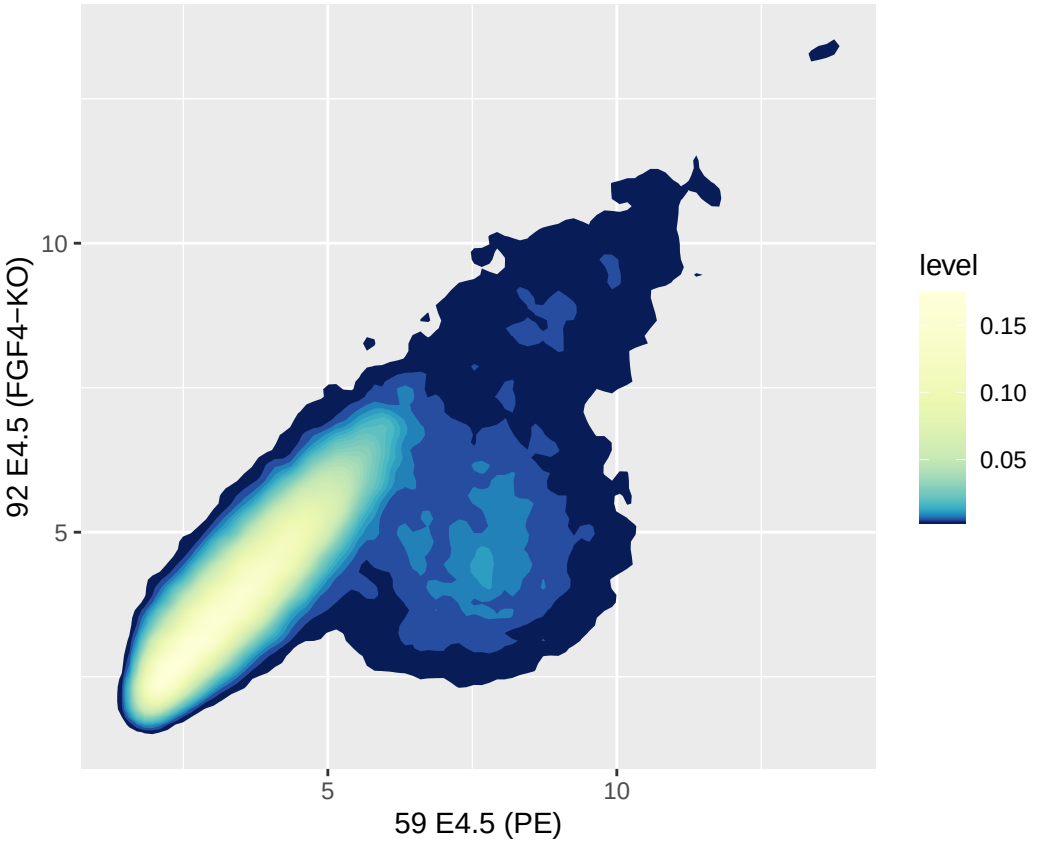
o avoid possible loss of information we can adjust the bandwidth and binning parameters of the `geom_density2d` function.

For the color representation we use the `RColorBrewer` package.

```
library(RColorBrewer)

colorscale = scale_fill_gradientn(
    colors = rev(brewer.pal(9, "YlGnBu")),
    values = c(0, exp(seq(-5, 0, length.out = 100))))
```

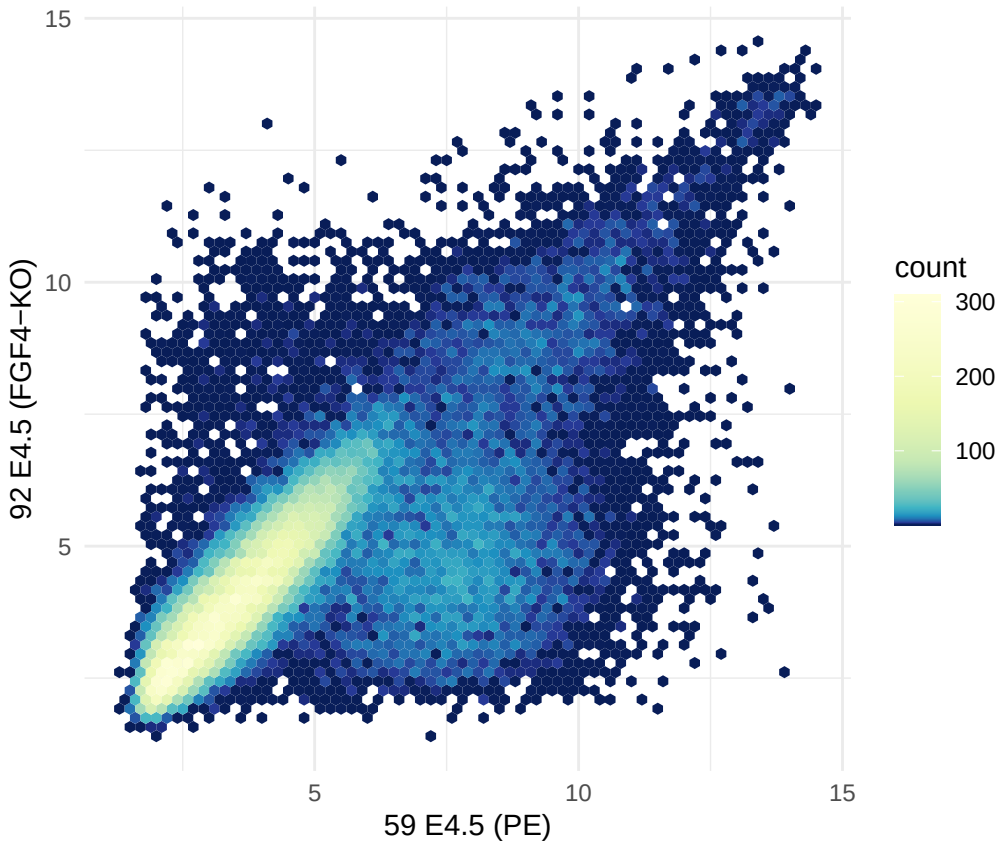
```
scp + stat_density2d(h = 0.5, bins = 60,
          aes( fill = ..level..), geom = "polygon") +
  colorscale + coord_fixed()
```



To get around all these problems, hexagonal binning can be used.

As a small hint: you can always change the default theme of ggplot2 and for example choose a more subtle one! Here I added `theme_minimal` which, in my opinion, improves the appearance of the plot.

```
scp + geom_hex(binwidth = c(0.2, 0.2)) + colorscale +
  coord_fixed() + theme_minimal()
```



Visualizing more than two dimensions

One possibility to display more than two dimensions are the following parameters:

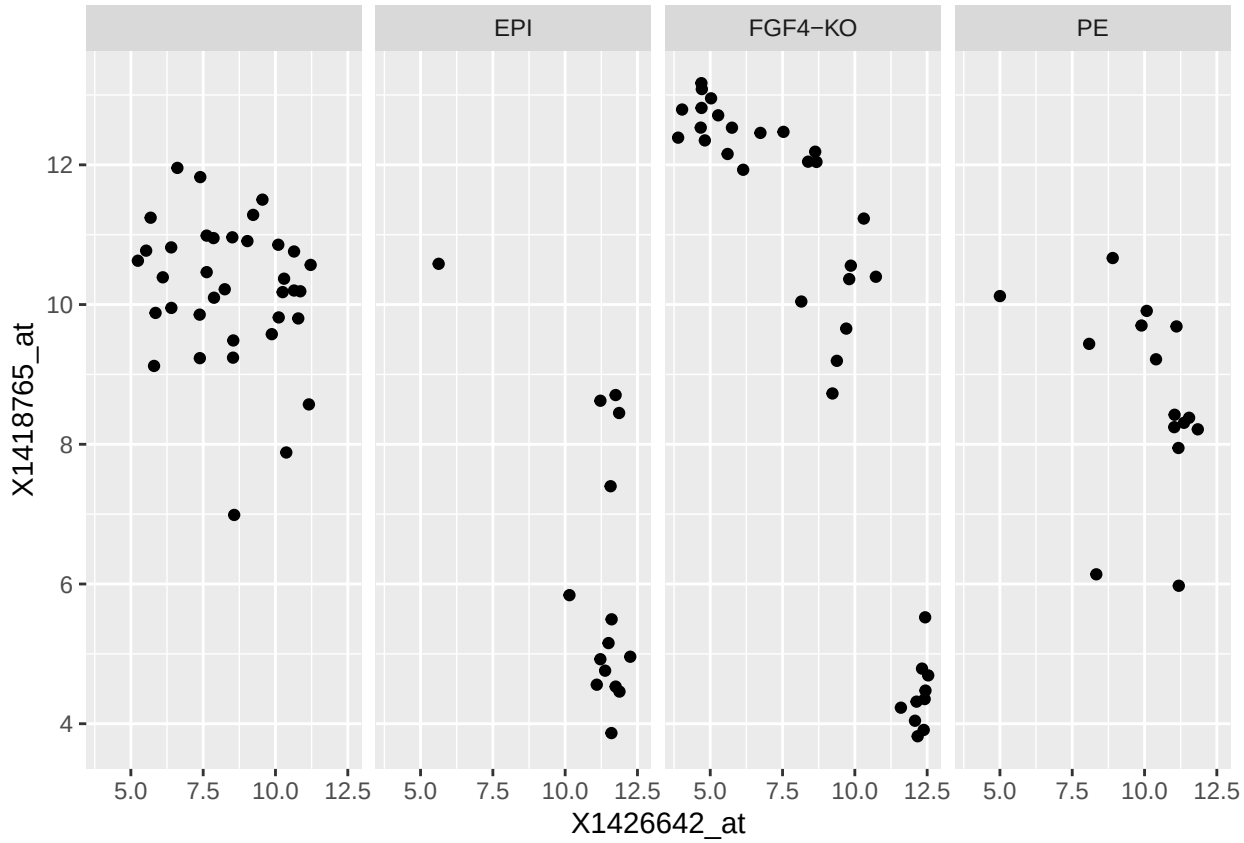
- fill
- color
- shape
- size
- alpha

Holmes and Huber (2018)

Normally, one would choose at most one or two of these parameters to represent additional dimensions. If this selection is not enough, you can use faceting.

Faceting

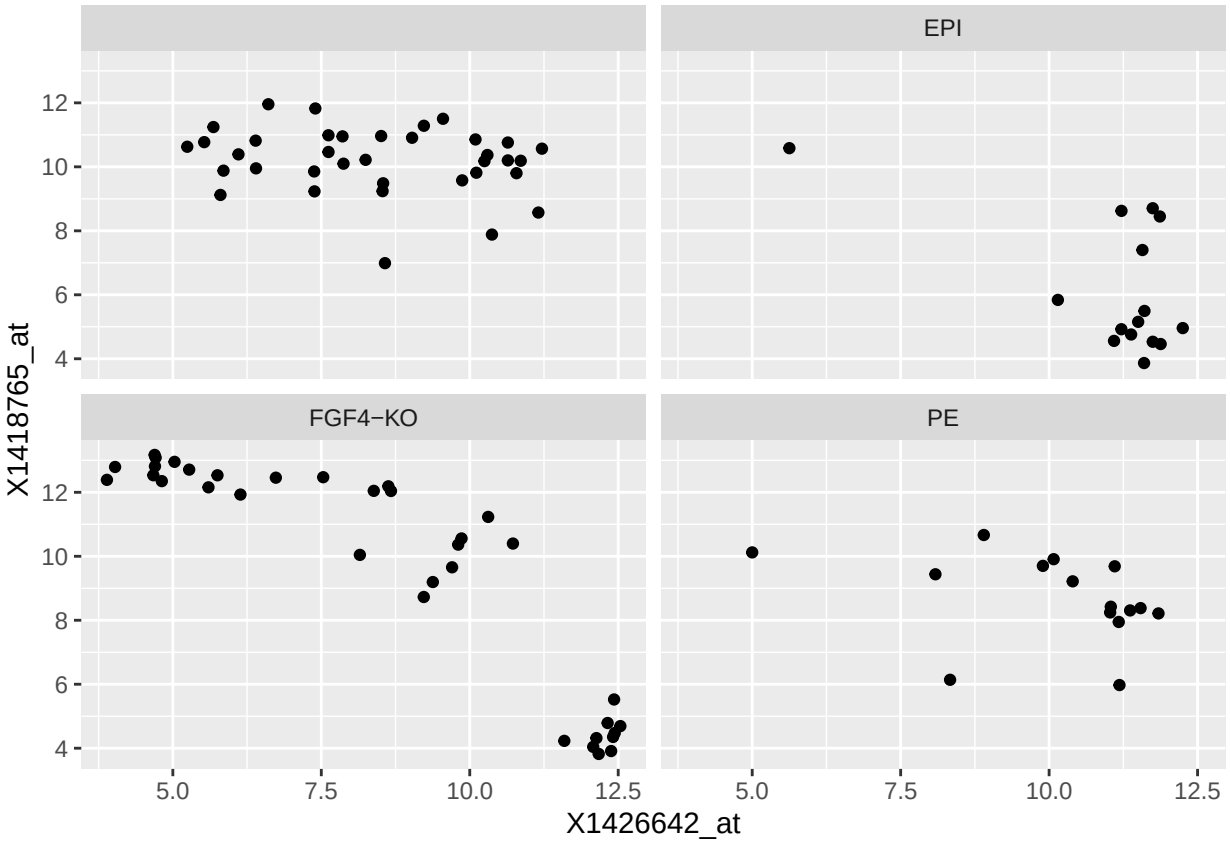
```
ggplot(dftx, aes( x = X1426642_at, y = X1418765_at)) +
  geom_point() + facet_grid( . ~ lineage )
```



Exercise:

Try displaying the facet plot in two rows using the `facet_wrap` function!

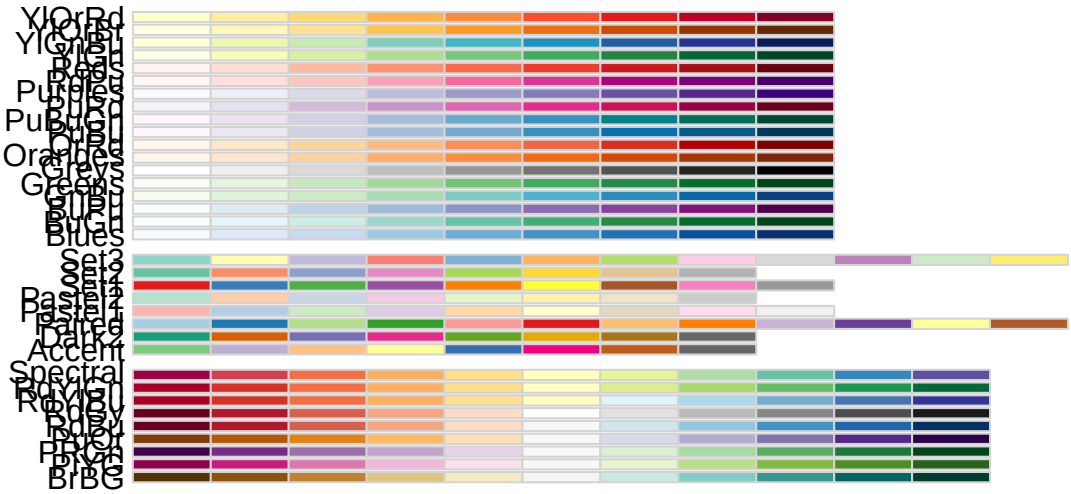
```
ggplot(dftx, aes( x = X1426642_at, y = X1418765_at)) +
  geom_point() + facet_wrap( . ~ lineage, nrow=2 )
```



Colors

To get an overview of the available and visually appealing color schemes, we can again use the `RColorBrewe` package.

```
display.brewer.all()
```



We can get additional information on each of the color scales (e.g. with regard to colorblind-friendliness).

```
head(brewer.pal.info)
```

```
##      maxcolors category colorblind
## BrBG        11      div       TRUE
## PiYG        11      div       TRUE
## PRGn        11      div       TRUE
## PuOr        11      div       TRUE
## RdBu        11      div       TRUE
## RdGy        11      div      FALSE
```

We get the appropriate colors by using the `brewer.pal` function, specifying the number of colors needed in the first argument and selecting the color palette in the second.

```
brewer.pal(4, "RdYlGn")
```

```
## [1] "#D7191C" "#FDAE61" "#A6D96A" "#1A9641"
```

Heatmaps

If we want to represent a large matrix, such as the expression of multiple cells across multiple genes, heatmaps are an appropriate form of visualization. A good package for displaying such heatmaps is `pheatmap`.

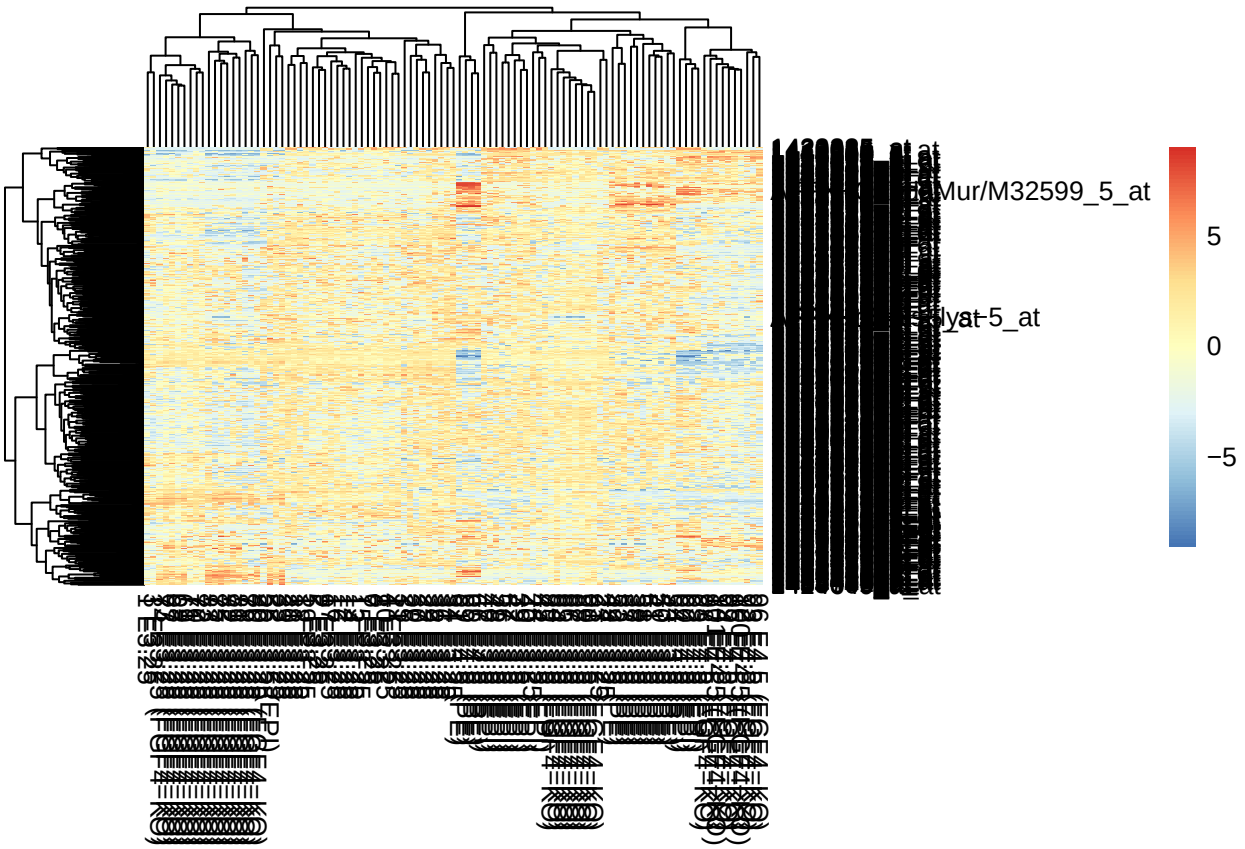
```
library("pheatmap")
```

First, we prepare our data set. We use the `rowVars` function to calcualte the variance of each row (gene) and order the genes by decreasing variance. Then, we select the first 500 most variable genes. Next, we write a function that normalizes the data and call it `rowCenter`.

```
topGenes = order(rowVars(Biobase::exprs(x)), decreasing = TRUE)[1:500]
rowCenter = function(x) { x - rowMeans(x) }
```

As the first argument to the `pheatmap` function, we specify the normalized and extracted genes as our data set.

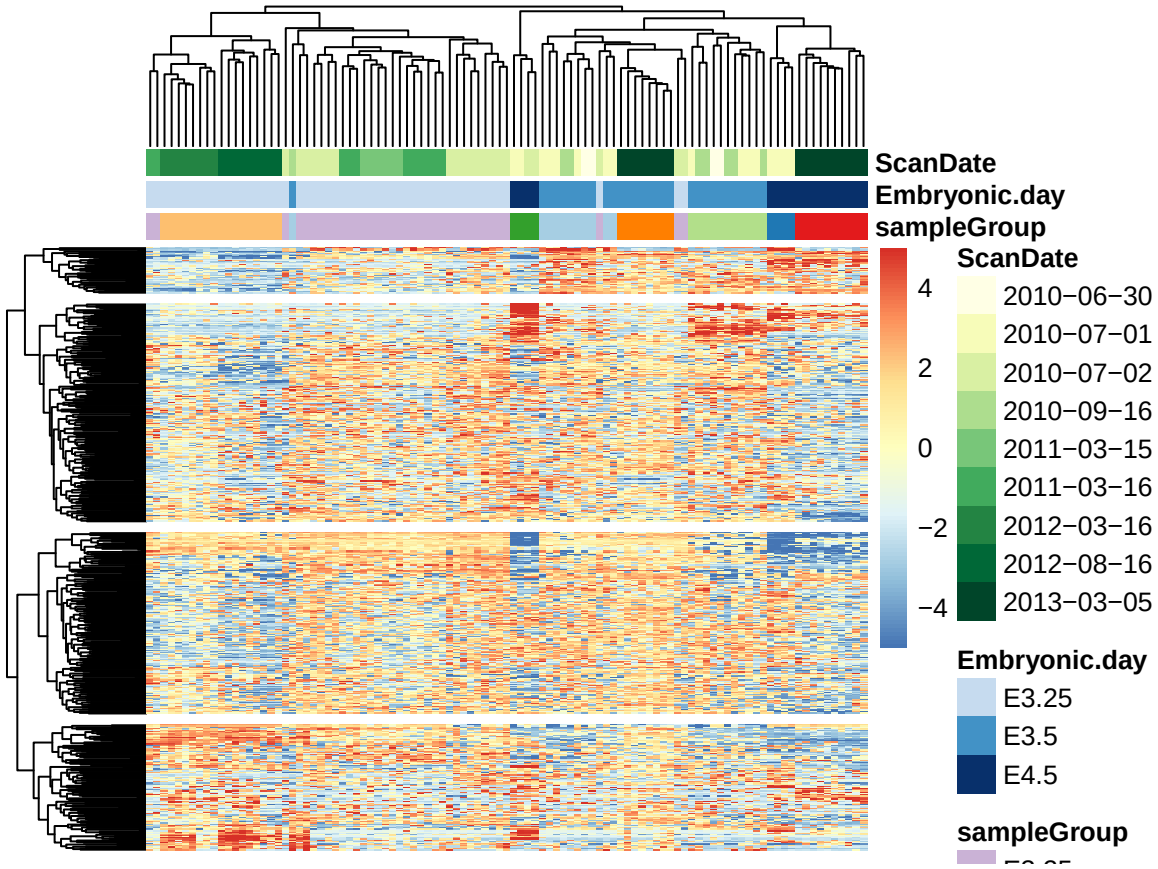
```
pheatmap( rowCenter(Biobase::exprs(x)[ topGenes, ] ) )
```



If we use the default settings of the function our heatmap looks very messy. So we should adjust some parameters to make it more visually appealing.

```
pheatmap( rowCenter(Biobase::exprs(x)[ topGenes, ] ),
  show_rownames = FALSE, show_colnames = FALSE,
  breaks = seq(-5, +5, length = 101),
  annotation_col =
    pData(x)[, c("sampleGroup", "Embryonic.day", "ScanDate") ],
  annotation_colors = list(
    sampleGroup = groupColor,
```

```
    genotype = c(`FGF4-KO` = "chocolate1", `WT` = "azure2"),
    Embryonic.day = setNames(brewer.pal(9, "Blues")[c(3, 6, 9)],
                             c("E3.25", "E3.5", "E4.5")),
    ScanDate = setNames(brewer.pal(nlevels(x$ScanDate), "YlGn"),
                        levels(x$ScanDate))
  ),
  cutree_rows = 4
)
```



Further reading

Session info

```
## R version 4.0.2 (2020-06-22)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS  10.16
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
```

```
##
## attached base packages:
## [1] stats4    parallel  stats     graphics  grDevices utils     datasets
## [8] methods   base
##
## other attached packages:
##  [1] pheatmap_1.0.12      Hmisc_4.4-2          Formula_1.2-4
##  [4] survival_3.2-7       reshape2_1.4.4       dplyr_1.0.2
##  [7] ggplot2_3.3.3        Hiiragi2013_1.24.0   xtable_1.8-4
## [10] RColorBrewer_1.1-2   mouse4302.db_3.2.3   org.Mm.eg.db_3.11.4
## [13] MASS_7.3-53          KEGGREST_1.28.0      gtools_3.8.2
## [16] gplots_3.1.1         geneplotter_1.66.0   annotate_1.66.0
## [19] XML_3.99-0.5         AnnotationDbi_1.50.3 IRanges_2.22.2
## [22] S4Vectors_0.26.1     lattice_0.20-41      genefilter_1.70.0
## [25] cluster_2.1.0        clue_0.3-58          boot_1.3-25
## [28] affy_1.66.0          Biobase_2.48.0       BiocGenerics_0.34.0
##
## loaded via a namespace (and not attached):
##  [1] nlme_3.1-151          bitops_1.0-6          bit64_4.0.5
##  [4] httr_1.4.2            backports_1.2.1       tools_4.0.2
##  [7] utf8_1.1.4            R6_2.5.0              affyio_1.58.0
## [10] rpart_4.1-15          KernSmooth_2.23-18    DBI_1.1.0
## [13] mgcv_1.8-33           colorspace_2.0-0      nnet_7.3-14
## [16] withr_2.3.0           gridExtra_2.3         tidyselect_1.1.0
## [19] bit_4.0.4             compiler_4.0.2        preprocessCore_1.50.0
## [22] cli_2.2.0             htmlTable_2.1.0       isoband_0.2.3
## [25] labeling_0.4.2        checkmate_2.0.0       caTools_1.18.0
## [28] scales_1.1.1          hexbin_1.28.1         stringr_1.4.0
## [31] digest_0.6.27         foreign_0.8-81        rmarkdown_2.6
## [34] XVector_0.28.0        base64enc_0.1-3       jpeg_0.1-8.1
## [37] pkgconfig_2.0.3       htmltools_0.5.0       htmlwidgets_1.5.3
## [40] rlang_0.4.10          rstudioapi_0.13       RSQLite_2.2.1
## [43] generics_0.1.0        farver_2.0.3          RCurl_1.98-1.2
## [46] magrittr_2.0.1        Matrix_1.3-0          Rcpp_1.0.5
## [49] munsell_0.5.0         fansi_0.4.1           lifecycle_0.2.0
## [52] stringi_1.5.3         yaml_2.2.1            zlibbioc_1.34.0
## [55] plyr_1.8.6            grid_4.0.2            blob_1.2.1
## [58] crayon_1.3.4          Biostrings_2.56.0     splines_4.0.2
## [61] knitr_1.30            pillar_1.4.7          glue_1.4.2
## [64] evaluate_0.14         latticeExtra_0.6-29   data.table_1.13.6
## [67] BiocManager_1.30.10   png_0.1-7             vctrs_0.3.6
## [70] gtable_0.3.0          purrr_0.3.4           assertthat_0.2.1
## [73] xfun_0.19             tibble_3.0.4          memoise_1.1.0
## [76] ellipsis_0.3.1
```


Holmes, Susan, and Wolfgang Huber. 2018. *Modern Statistics for Modern Biology.* Cambridge University Press.

Wilke, Claus O. 2019. *Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures.* O'Reilly Media.